A Virtual Assembler for an Abstract Machine

Design and implementation of an incremental and retargetable
code generator for term rewriting systems

Leon Moonen

A Virtual Assembler for an Abstract Machine

Design and implementation of an incremental and retargetable
code generator for term rewriting systems

Opdrachtgevers: Centrum voor Wiskunde en Informatica (CWI), afdeling Programmatuur,
Hogeschool Eindhoven, Faculteit Techniek, Studierichting Hogere Informatica

Mentoren: prof. dr. P. Klint, CWI, afdeling Programmatuur, groepsleider AP3
dr. H.R. Walters, CWI, afdeling Programmatuur, senior researcher
P. van Baalen, Hogeschool Eindhoven, Studierichting Hogere Informatica

Auteur: Leon Moonen, Hogeschool Eindhoven, Studierichting Hogere Informatica

Plaats: Amsterdam
Datum: 6 juni, 1994

cover picture: Juggler of Toltec Origin, Pre-Hispanic Mexican drawing, circa 950 A.D.

Abstract

Term rewriting systems are frequently used as an implementation technique for algebraic specifications. We present an incremental and retargetable code generator for conditional term rewriting systems, which is used as the back end of an ASF+SDF compiler. The specifications are developed in the ASF+SDF Meta-environment and can be translated by the compiler into stand-alone programs. These are very convenient since they can be used without the (rather large) Meta-environment. Furthermore, they execute much faster than the ones that are interpreted within that environment.

The front end of the compiler translates an ASF+SDF specification into intermediate code called μARM. This code implements a term rewriting system on the Abstract Rewriting Machine, an abstract machine dedicated to rewriting. The code generator accepts a term rewriting system described in the μARM language and generates machine-specific assembly code that is functionally equivalent to the input.

Investigation of several existing tools and techniques for code generation, such as the use of a code generator generator or re-using the back end of an existing compiler led to the conclusion that these methods would not produce acceptable results. Therefore, we introduce our own approach: a *virtual assembler* for μARM. Our code generator can be outlined as follows: first translate a μARM program into Virtual Assembly (VA) code (the VA language can be thought of as a generic assembly language). Then, translate the VA code into actual machine-specific assembly code using the virtual assembler. We use a macro processor as virtual assembler since it is available on all target architectures and it can easily be parameterized using macro definitions. Consequently, the translation can easily be retargeted by changing the mapping specified in the macro definition file.

The generated code can be linked and executed together with the run-time system or with the hybrid μARM interpreter. The run-time system is a skeleton which implements the environment for the generated code; it implements the machine model of the abstract rewriting machine. The hybrid interpreter is an interpreter for μARM that can handle external (pre-compiled) function definitions in combination with function definitions in μARM that are interpreted. This allows one to compile stable parts and execute them in conjunction with interpreted parts, such that a high turn around time can be achieved for TRSs that are still under construction.

The generated code executes in the order of 500000 rewrite steps per second and a maximum of approximately 1000000 rewrite steps per second was measured. It is roughly twice as fast as ASF2C: a prototype ASF+SDF compiler; and about 100 times as fast as the interpreted ASF+SDF Meta-environment.

Preface

This thesis is the result of my graduation research project carried out at the Software Technology department of the Centre for Mathematics and Computer Science (CWI) from January 1994 until June 1994. The starting point of the project was the design and implementation of a code generator for the ASF+SDF compiler which is being developed at CWI.

I would like to thank Prof. dr. P. Klint who introduced me to the project, dr. H.R. Walters and drs. J.F.Th. Kamperman for guiding me with constructive criticism and giving useful suggestions on all aspects of my work. Furthermore, I would like to thank drs. E.A. van der Meulen and drs. E. Visser for providing me with a quick course in algebraic specification using ASF+SDF (i.e., parts of the courses "Programmeeromgevingen" and "Software Engineering" which are taught at the University of Amsterdam).

Paul Klint, Pum Walters, Pim van Baalen and Daniel Dekkers read draft versions of this document. Arie van Deursen and Jasper Kamperman assisted me with the conquering of LaTeX.

I would like to thank you all.

Leon Moonen
Amsterdam
June 1994

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we will describe the design and implementation of an incremental and retargetable code generator for term rewriting systems. A code generator is the last stage or back end of a compiler. It is used to generate (machine-specific) executable code. An *incremental* code generator is a code generator that is able to reuse stable parts and only recompile those parts that have been changed. Accordingly, the generated code is being adapted rather than regenerated from scratch. A code generator is called *retargetable* when it is relatively easy to change the architecture for which the code is generated (i.e., the target architecture).

1.1 Term rewriting systems and algebraic specifications

A term rewriting system consists of a set of replacement rules or *rewrite rules* wich describe a term rewriting process. Rewrite rules are equations that describe the replacement of one pattern (the left hand side or lhs) by another pattern (the right hand side or rhs). A rule may also contain some conditions that must hold for the rule to be applicable. Term rewriting is an iterative process which repeatedly attempts to recognize an instantiation of the left-hand side of a rewrite rule. This instantiation is called *redex* (*red*ucible *ex*pression). When a redex is found, the conditions of the corresponding rule are checked (which involves a recursive invocation of the rewriting process) and the established redex is replaced by the instantiation of the right-hand side of that rule. This process is also called the *reduction* of a term.

1.1.1 Algebraic Specifications

Term rewriting systems are frequently used as an implementation technique for *algebraic specifications*. Algebraic specification is a technique which can be used for the formal specification of a wide variety of problems. For example, algebraic specifications can be used for the definition of languages (for programming, for writing specifications, for querying databases, for text processing, or for dedicated applications). Furthermore, they can be used for the description and rapid prototyping of software systems.

Essential in algebraic specifications is that they are used to define *algebras*. An algebra consists of a set of values and a number of functions operating on these values. A well known example of an algebra is that of the natural numbers, with values 0,1,2,... and functions such as addition or multiplication.

An algebraic specification characterizes certain algebras. This is done by defining a *signature* (i.e., a set of functions with type information) and *equations* that are required to hold for the terms over that signature. For example, the signature for natural numbers can be specified as follows (`NAT` is the type of a natural number):

```
signature:
   z                   ->  NAT
   succ( NAT )         ->  NAT   % denotes the successor of a number
   plus( NAT, NAT )    ->  NAT   % denotes addition of two numbers
```

Using the constant and functions in this signature, infinitely many terms can be made. Examples are: `z`, `succ(z)`, `succ(succ(z))` and `plus(succ(z),succ(succ(z)))`. The following equations are required to hold in order to construct a normal form of a given term:

```
equations:
   plus( z, Nat )            =  Nat
   plus( s( Nat1 ), Nat2 )   =  plus( Nat1, s( Nat2 ) )
```

Using these equations, the term `plus(succ(z),succ(succ(z)))` can be reduced to the normal form `succ(succ(succ(z)))`.

ASF+SDF

The term rewriting systems for which code has to be generated are compiled out of an ASF+SDF specification. The ASF+SDF formalism [BHK89] is a general-purpose algebraic specification formalism which is designed at the Centre for Mathematics and Computer Science (CWI) and the University of Amsterdam. ASF+SDF is the combination of the two formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism). The ASF formalism [BHK87] is based on the notion of modules, each consisting of a *signature* defining the abstract syntax of functions and a set of *conditional equations* defining their semantics. The SDF formalism [HHKR89] allows the simultaneous definition of concrete (i.e., lexical and context-free) and abstract syntax and implicitly defines a translation from text strings to abstract syntax trees. These are trees in which each node represents an operator (function) and the children of that node represent the operands (parameters) of that operator. Figure 1.1 gives an example of the abstract syntax tree for the string "$9 - 5 + 2$".

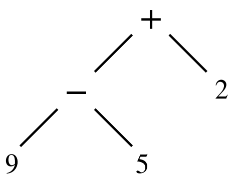


Figure 1.1 The abstract syntax tree for $9 - 5 + 2$.

The main idea of ASF+SDF is to identify the abstract syntax defined by the signature in ASF specifications with the abstract syntax defined implicitly by an SDF specification. This results in a standard mapping from text to abstract syntax tree which gives the possibility to associate semantics with (the tree representation of) text and to introduce user-defined notation in specifications. ASF+SDF is therefore a modular specification formalism for the integrated definition of syntax and semantics.

The Meta-environment

ASF+SDF specifications can be executed using the ASF+SDF Meta-environment [Kli93]. This is an development environment for the automatic generating of interactive environments for manipulating programs, specifications, or other texts written in the ASF+SDF language. The generation process is controlled by a definition of the target language, which typically includes features as syntax, pretty printing, type checking and execution of programs. The SDF part of a specification is used to generate *lazy incremental parsers* [Rek92] and *syntax directed editors* [Koo92] for the language defined. The ASF part of a specification is *interpreted* as a term rewriting system which can be used to *reduce* terms in this language [Wal91].

1.2 A compiler for ASF+SDF

The code generator described in this thesis is part of a compiler for ASF+SDF specifications which is under construction at CWI. This compiler converts an ASF+SDF specification into a *stand-alone program*. Stand-alone executable specifications are very convenient since they can be used without the (rather large) Meta-environment. Furthermore, they execute much faster than the ones that are interpreted within the Meta-environment.

1.2.1 Front End

The front end of the compiler translates an ASF+SDF specification into intermediate code called μARM. This intermediate code implements a term rewriting system on the *abstract rewriting machine* [KW93].

The abstract rewriting machine

The abstract rewriting machine is an imaginary machine dedicated to term rewriting. The machine has three stacks, a heap and several internal registers. It performs an iterative rewriting process in which it is guided by the μARM program.

The programming language μARM has control structures for analyzing terms, recursively calling the rewrite process, and generating new terms. These control structures use and manipulate the machine's registers, stacks and its heap. There are no instructions to manipulate these memories explicitly, nor can arithmetic or logic computations be performed, since μARM only concerns rewriting.

The cyclic process can be described as follows. Initially all function symbols in a subject term are pushed in pre-order on a stack (called the control stack), which for that moment may be regarded as the input stack. In general, no tree structures will be made by μARM for terms which have not yet been processed. Consequently, an actual tree representation is made only if a (sub-)term is a normal form.

Each cycle a symbol is taken from the control stack and is interpreted as the outermost function symbol of a to-be-rewritten term. The immediate sub-terms of this term are found (that is, if the function symbol is not a constant) on a second stack, the argument stack. As required, the sub-terms are analyzed further, until a rewrite rule matches. Then conditions are checked and the corresponding right-hand side is substituted. This is done by pushing the appropriate symbols (and sub-terms of terms on the argument stack) onto the control stack.

If no rewrite rule matches, a normal form is constructed from the function symbol on the control stack and the required number of terms from the argument stack.

Execution of this process continues until the control stack is empty, at which time the argument stack holds the normal form of the input term.

1.2.2 Back End

The back end or code generator that is presented in this thesis accepts a term rewriting system described in the μARM language and generates machine-specific code that is functionally equivalent to the input. The generated code can be linked together with the *run-time system* or with the *hybrid* μARM *interpreter*. The run-time system is a skeleton which implements the environment for the generated code. I.e., it implements the machine model of the abstract rewriting machine. The hybrid interpreter is an interpreter for μARM that can handle external (pre-compiled) function definitions in combination with function definitions in μARM that are interpreted. This allows one to compile stable parts and execute them in conjunction with interpreted parts, such that a high turn around time can be achieved for TRSs that are still under construction.

1.3 Thesis overview

The project started with research to existing tools and techniques that can be used for code generation, such as the use of a code generator generator (a program that generates back ends) or re-using the back end of an existing compiler. An overview of these techniques is presented in Chapter 2 on page 9. A detailed discussion of code generator generators can be found in Appendix D on page 63. The re-use of existing back ends is discussed in more detail in Appendix E on page 69

In our case, these tools and techniques result in a suboptimal solution. Therefore we will introduce our own approach: a *virtual assembler* for μARM. This approach is presented in Chapter 2. A large part of the code generator is designed and implemented using the ASF+SDF algebraic specification technique. Consequently, it can be used to create a stand-alone executable version of itself. The design and implementation are discussed in Chapter 3 on page 13. The complete specification is presented in Appendix G on page 77.

In Chapter 4 on page 21 our approach is evaluated and compared with the Meta-environment, the μARM interpreter and the previous ASF2C compiler [KW93]. In Chapter 5 on page 25 we will formulate some conclusions and give some directions for further

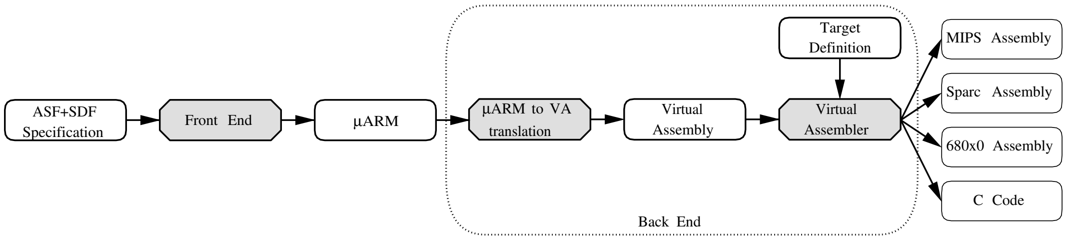


Figure 1.2 Overview of the ASF+SDF compiler.

research.

A comprehensive overview of the ASF+SDF formalism and the Meta-environment can be found in Appendix A on page 39. The front end of the ASF+SDF compiler is described in Appendix C on page 55. The Abstract Rewriting Machine and its programming language (μARM) are discussed in greater detail in Appendix B on page 47.

The source language of the virtual assembler, the *virtual assembly language*, is described in Appendix F on page 73. A possible conversion of this language into C code is presented in Appendix H on page 91.

Appendix I on page 93 presents an example of the translation of an ASF+SDF specification into C functions that can be linked together with the run-time system or with the hybrid interpreter to create a stand-alone term rewriting system.

1.4 Terminology

The technical terms and phrases that are used are described in the glossary on page 27.

In this thesis, we will use the abbreviation *"trie"* (pronounced as try) to denote a (part of a) μARM program. Trie is the abbreviation for Term Recognizing and Instantiating Entity. A trie contains one or more μARM statements and is typically used to denote a composition of instructions that can be treated as one instruction (for example the instructions that have to be executed when a match succeeds).

Furthermore, we will use the phrases *"assembly language"* and *"machine language"* interchangeably to denote the low level mnemonic code that is usually translated into object code by an assembler.

1.5 The research institute CWI

The work described here was carried out at the Software Technology department of the Centre for Mathematics and Computer Science (CWI). The CWI is an independent research institute active in the fields of mathematics and computer science. CWI is part of the Mathematical Centre Foundation, a non-profit organisation founded in 1946, aiming to foster mathematics and computer science in the Netherlands.

1.5.1 Mission

The mission of CWI is twofold:

- to perform frontier research in mathematics and computer science;
- to transfer new knowledge in these fields to society in general and to trade and industry in particular.

1.5.2 Frontier research

CWI research activities are at the frontiers of science. Fundamental contributions are made to the progress of mathematics and computer science. At the same time, much attention is payed to the applicability of the results of the research. Strategic considerations and applicability play an important role in the selection of new projects.

The CWI has many contacts with organisations that are involved in the translation of results of scientific research into practical applications. Examples of such organisations are the Dutch Organisation for Applied Research (TNO) and other technological institutes like ECN, NLR, and MARIN. In several projects there is co-operation with these institutes for applied research.

1.5.3 Knowledge transfer and contract research

Results of research carried out at CWI are made available in publications and reports. In addition, results are presented at scientific conferences and advanced short courses are given. Knowledge transfer also takes place through the mobility of scientific staff, either in the form of exchange of personnel with, or transfer to, universities, research institutes or industry. The execution of contract research is considered to be an important form of knowledge transfer and interaction with society generates new areas of investigation.

1.5.4 Relations with university research

Many senior scientists at CWI hold part-time positions as university professors. As a result, an extensive network of personal contacts exists, which forms the basis for co-operation with university research groups.

1.5.5 Finances

CWI receives a subsidy from NWO, the National Organisation for Scientific Research, for performing fundamental research. This subsidy accounts for 70 percent of the total income of the institute. The remaining 30 percent is obtained through national and international research programmes (e.g., ESPRIT, RACE, BRITE) and through contract research commissioned by industry.

1.5.6 Departments

CWI has seven scientific departments. About half of these are specialized in mathematical subjects, the other half does computer science research. These departments are:

AA: Algorithmics and Architecture

AP: Software Technology

CST: Computer-systems en Telematics

IS: Interactive Systems

AM: Analysis Algebra and Geometry

BS: Operations Research Statistics & System Theory

NW: Numerical Mathematics

Furthermore, CWI has its own Library & Information Service.

1.5.7 Department of Software Technology

The department of Software Technology performs research in areas relevant for large scale software engineering. This includes the design of programming languages, methods and environments and, investigation of techniques for parallel computing. The department is divided into the following groups:

AP1: Semantics

AP2: Concurrency and real-time systems

AP3: Extensible programming environments

AP4: Algebraic and syntactic methods

AP5: Logic and language

The head of the department is Prof.dr. J.W. de Bakker

The work described in this thesis was carried out in group AP3. This group performs research on programming environments and program generation. This includes the following subjects:

- incremental program generation (e.g., incremental generation of type checkers and evaluators from formal semantics definitions).
- generation of interactive programming environments.
- generation of compilers for parallel computers (the ESPRIT-COMPARE project).

The group leader is Prof.dr. P.Klint.

Chapter 2

Code generation for ASF+SDF

The input for the back end is code which implements a term rewriting system on the abstract rewriting machine (ARM). The abstract rewriting machine and its programming language are discussed in Appendix B on page 47. The output of the back end is assembly language code or C code that is functionally equivalent to the input. We aim at several different target architectures (e.g., Sparc, MIPS and 680x0). For portability and easy testing, we also include C code as a target.

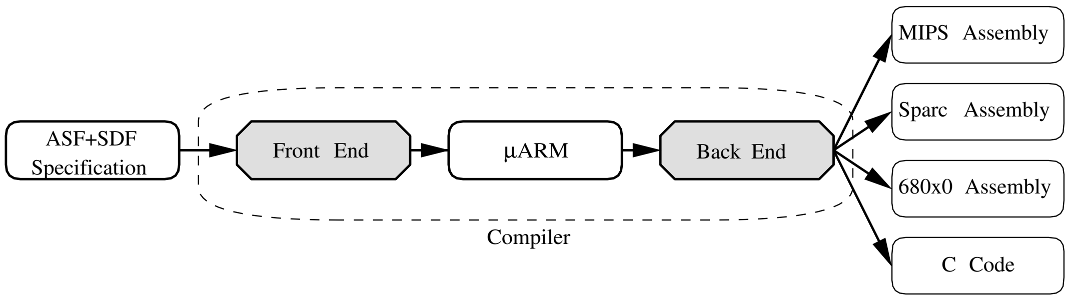


Figure 2.1 The ASF+SDF compiler.

An important requirement on the back end is that it should support an acceptable degree of *incrementality*. It should have the ability to reuse stable parts and only re-compile those parts that have been changed by the user or the front end. Incrementality is essential for large scale software development since it significantly reduces the *turn around time*. This is the time between making a change in the specification and being able to use the newly compiled code.

In a compiler, several degrees of incrementality (or *grain sizes*) are possible. The degree of incrementality is a measurement for the size of the smallest change that is passed through to the back end. Examples of grain sizes are per statement and per function. We choose for the back end to support incrementality per function definition. A rationale for this choice is given in the remainder of this section.

The front end supports two degrees of incrementality: many changes to a specification result in re-compilation of a single function definition; only certain changes require re-compilation of several function definitions (similar to the behaviour of the 'make' facility that can be found in most operating systems). Accordingly, the smallest degree of incrementality supported by the front end is a single function definition. As a result, the smallest degree of incrementality that can be supported by the back end is also a single function definition.

The ASF+SDF compiler is intended to be used in two ways:

1. in cooperation with the μARM interpreter. In the context of large scale software development it is convenient to be able to compile the stable parts of the software but to process the parts still under development as fast as possible. For this reason a *hybrid* μARM interpreter which can cooperate with compiled parts is provided.

2. stand-alone, in combination with a run-time system. This is a system which implements the skeleton of the machine model for the abstract rewriting machine. The run-time system is basically the same as the μARM interpreter when it does not use any interpreted functions.

The back end should deliver code which cooperates with the interpreter. Given the design of the interpreter, a convenient interface to use is that of *external functions*. This is a mechanism intended for importing functions that are used, but not defined in a specification. This mechanism is equally suitable to replace interpreted functions by their compiled counterpart. The imported functions receive their definition by linking the external function code to the interpreter (a planned extension is to allow dynamic loading of the external functions to the interpreter). Since the run-time system is designed as a subset of the interpreter it uses the same calling conventions so the same functions can be used. Consequently, a convenient degree of incrementality to be supported by the back end is a single function definition.

In the remainder of this chapter we will discuss various methods of machine-specific code generation and we will introduce our approach: a *virtual assembler*.

2.1 Code generation using existing tools and techniques

In this section we will discuss two methods that make use of existing tools and techniques for the generation of machine-specific code. The first approach concerns the use of tools and techniques to generate a back end for the given input language. It uses a so called *code generator generator* or *back end generator*. The other approach is to re-use the back end of a compiler that was originally developed for another language. This is discussed in Section 2.1.2 on page 11.

2.1.1 Generate a back end with a code generator generator

A code generator generator is a system that builds a *tree pattern matcher* out of a declarative *code generator description*. A tree pattern matcher is a program which analyses a tree and performs actions (e.g., it generates code) when a certain pattern is discovered. A code generator description is a "database" holding the machine language commands that correspond to constructs of the input language. The generated pattern matcher can be used to find the optimal machine language code for a given input program. The code generator description (GCD) for a given architecture can easily be ported to another target architecture so it is relatively easy to build portable back ends.

We have investigated three back end generators (see Appendix D on page 63):

- `BEG` ([ESL89, Emm90])
- `BURG` ([FHP92b])

- `iburg` ([FHP92a])

The major drawback is that we do not expect that any real optimizations can be done by these code generators since μARM is closely related to machine language. Another drawback is the effort needed to write a GCD. One has to design a CGD for every source language that has to be translated. This CGD can easily be ported to other architectures. Given the limited time frame of our project, the definition of a CGD which translates μARM is not feasible. Accordingly, we feel that the use of a code generator generator has little or no benefits in our specific case.

2.1.2 Use an existing back end

The idea behind re-using the back end of an existing compiler is that most well designed compilers consist primarily of two parts:

1. a language dependent and target independent front end which translates the source code into some kind of *intermediate representation (IR)*.
2. a (fairly) language independent and target dependent back end which translates the intermediate representation into actual machine code.

When μARM code is translated into the intermediate representation that is used by a given compiler we can use its back end(s) for code generation. This is a vary powerful scheme if there is a portable compiler available that has an intermediate representation which fits acceptably to μARM since all present and future ports of that compiler can be used for the code generation of our language.

After investigating several compilers (e.g., `GNU CC`, `lcc`, `VPCC` and `SML\NJ`) we can conclude that most of the back ends are especially designed to process an intermediate language derived from the "original" source language (in this case C and ML). Consequently, some difficulties may arise during the translation of μARM code into this intermediate language. Furthermore, we expect inefficiencies since the relatively low level μARM code must be translated to a higher level intermediate language before it can be translated to a low level machine language. Accordingly, we will not use these back ends.

The Amsterdam Compiler Kit (ACK) [TvSKS83b] is a collection of programs designed to simplify the task of producing portable (cross) compilers and interpreters. For each input language (in our case μARM) a "front end" must be written to translate it into a common intermediate representation called EM. ACK contains back ends that can translate this intermediate code into assembly language. A drawback of this approach is that there are no back ends for MIPS architectures available and the SPARC back end produces rather slow code. Consequently, we will not use ACK.

The Amsterdam Compiler Kit and the back ends described above are discussed in more detail in Appendix E on page 69.

2.2 Code generation using a virtual assembler

We present a scheme which uses a *virtual assembler* to translate μARM code into machine-specific assembly language. It can be outlined as follows:

1. define a *virtual assembly language* which can easily be ported to different target architectures.

2. develop a translation from μARM code into this virtual assembly language.
3. develop a mapping from the virtual assembly language on the actual assembly languages using a parameterizable *virtual assembler*.
4. find a *native assembler* for the translation from the actual assembly language into object code. A native assembler is an assembler that is designed for, and implemented on, a specific target architecture.

This scheme has several advantages over the direct translation from μARM code to machine-specific assembly language:

- only step 3 and step 4 are machine-specific and need to be done for every target architecture.
- step 4 requires a tool which is in general publicly available for a given architecture.
- step 3, the only remaining machine-specific part, is a trivial translation which can be done by macro expansion using a preprocessor.

Consequently, the only machine-specific knowledge we have to deal with within this scheme is hidden in the definition of the mapping used in step 3. Only this mapping needs to be redefined to retarget the compiler to another architecture. When the virtual assembly language is kept comparatively small, the scheme presented here is a powerful approach to both portable and retargetable code generation in our situation.

Chapter 3

A virtual assembler for μARM

There are two properties one likes to see in a translation process that is frequently going to be used in the context of large scale software engineering:

- The translation should be easily *retargetable.* It must be relatively easy to change the target architecture for which machine code is generated.
- The translation should be easily *portable.* It must be relatively easy to change the architecture on which the translation is executed (the host architecture).

These properties have a great influence on the usability of the translation process, so in general it is a good idea to add them to the list of other requirements (such as, in our case, incremental processing and the ability to cooperate with the hybrid interpreter).

Easy portability can be achieved by eliminating all host architecture dependencies and by using of standardized tools that are publicly available for the target architectures.

Easy retargetability can be achieved by dividing the translation process into two *separate parts* [ASU86]:

1. a machine-independent "front end" which translates the input code into an (also machine-independent) intermediate representation (IR).
2. a machine-dependent code generator which translates this intermediate representation into machine-specific code.

It is important to keep the code generator as simple as possible since it is the part that needs to be build for every target architecture. Furthermore, it is convenient to concentrate all target dependencies in a, preferably parameterizable, data base that is easy portable

We propose a scheme that uses a *virtual assembly language* (VA) as intermediate language and a *virtual assembler* as code generator. The virtual assembler is parameterizable using a *definitions file* containing all target architecture dependencies.

The virtual assembly language is a generic and machine-independent counterpart of ordinary assembly languages. We feel that it is convenient in our case to abstract very little from the machine level, since the μARM language is so closely related to machine code. Moreover, when a higher abstraction level is used, inefficiencies may arise since the relatively low level μARM is translated into a higher level intermediate language and back to a low level machine language.

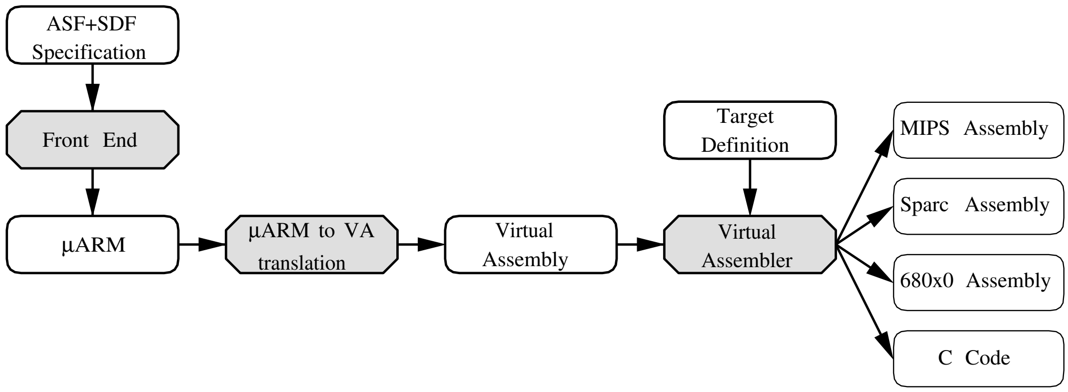


Figure 3.1 Overview of code generation for ASF+SDF using a virtual assembler.

The virtual assembler intermediate level is necessary since μARM instructions contain implicit flow of control and explicit flow of control is needed for assembler generation. The μARM to VA translation adds this information explicitly to the code.

The translation of μARM code into machine-specific code in our scheme can be outlined as follows (see also Figure 3.1):

1. translate μARM code into the virtual assembly language.
2. convert the virtual assembly language into machine-specific code or C code using the *virtual assembler*.
3. translate the generated assembly code into object code using a *native assembler* or C compiler.

These steps are discussed in more detail below, after the definition of the virtual assembly language.

3.1 Definition of a virtual assembly language

The following requirements must be taken into account for the definition of the virtual assembly language (VA):

a) Every μARM statement must be expressible as one (or a few) VA statement(s).

b) VA must be based on the same machine model as μARM (see Section B.2 on page 48 for a discussion of the μARM machine model).

c) VA must offer facilities to cooperate with the hybrid interpreter.

d) VA must be as close to "generic" assembly language as possible.

e) VA must contain **no** target machine dependencies.

f) The VA instruction set must be reasonable small to facilitate easy retargeting.

Requirement *a* and *b* originate from the fact that the output of the code generation process must be functionally equivalent to the input. The last three requirements (*d*, *e* and *f*) assure that the translation is easily retargetable. Note that incrementality imposes no additional requirements on the virtual assembly language.

The μARM instruction set (see Section B.3 on page 52) can be divided into four groups of functionally related instructions:

1. Instructions for unconditional flow of control.
2. Instructions for matching.
3. Instructions for stack manipulation.
4. Instructions that build nodes on the heap to represent (sub-)terms.

Since all μARM statements must be expressible in VA and since μARM and VA both operate on the same machine model, these groups recur, in one way or another, in the set of VA instructions:

1. **Instructions for unconditional flow of control:** These instructions are similar in VA to those in μARM. There is an instruction to restart the μARM rewriting cycle and an instruction for continuation with a given function symbol.

2. **Instructions for matching:** These instructions are somewhat different in VA since there are no target architectures that directly supports matching. Furthermore, the μARM matching instructions have an implicit conditional flow of control while it is necessary to have explicit flow of control for the generation of machine code.

   The VA instructions for matching are structured similar to the C `switch` statement. Note however that the syntax is slightly different since unique labels are added to the `case` and `default` instructions to denote the explicit flow of control.

3. **Instructions for stack manipulation:** In μARM, these instructions have the stacks to operate on encoded in their name. This results in several instructions having the same semantics but operating on a different stack (e.g., `tdrop` and `adrop`). In VA, the stacks to operate on are explicitly passed as arguments.

   Note that some μARM instructions have semantics that imply several stack manipulation instructions (e.g., `skip` and `retract`). These instructions are decomposed into several "atomic" VA instructions.

4. **Instructions to build nodes:** In μARM, these instructions have the stacks to operate on encoded in their name. In VA the stacks to operate on are explicitly passed as arguments. Furthermore, the μARM instructions have a statically known argument (α) which denotes the number of children of the node-to-be-build (either 0, 1 or 2). Depending on the value of α, functionally different code needs to be generated. This may result in inefficiencies when a *naive code generator* is used. A naive code generator is a code generator that emits code for a given (intermediate) instruction without optimizing the generated code. Such a code generator would generate a `switch` on the value of α. Since this value is statically known, it is also known which part of the switch statement is going to be executed. Consequently, the emitted code is suboptimal. In VA this is solved by encoding the value of α in the instruction name. Consequently, there are three VA instructions each handling a specific value of α. This allows a naive code generator to emit only the code that is actually used.

In the interpreter and in the run-time system, there are three names that are related to a μARM function symbol:

I the string literal which is the same as the μARM function symbol.

II the name of the function that performs the rewriting process of a given function symbol.

III the name of the variable which holds a reference to this function. This variable is used internally instead of the function symbol to allow for a fast association between a function symbol and its corresponding function.

The VA language contains some special instructions to declare which function and variable correspond to a given function symbol. Furthermore, it contains special instructions to define functions.

Group	*Instructions*

declarations	`DEFINE( FunSym, Function, FunVar, Id )`
	`FUNCTION( FunName )`
	`END`

flow of control	`RECYCLE` `GOTO( FunSym )`

matching	`SWITCH`
	`CASE( ThisLabel, Sym, NextLabel )`
	`DEFAULT( ThisLabel )`
	`PAIRSKIP`

stack manipulation	`PUSH( Stack )`
	`TOP( Stack )`
	`POP( Stack )`
	`MV( Val, Dest )`
	`NTH( Stack, N )`
	`DROP( Stack, N )`

building nodes	`BUILD0( FunSym, Dest )`
	`BUILD1( FunSym, Src, Dest )`
	`BUILD12( FunSym, Src, Dest )`
	`BUILD21( FunSym, Src, Dest )`

Table 3.1 Overview of the VA instruction set

The VA instruction set is presented in Table 3.1. These instructions are discussed in greater detail in Appendix F on page 73.

3.2 Translation of μARM into VA

The translation of μARM into VA involves two passes:

1. In the first pass, a unique ID is generated for every function symbol that is defined in the μARM program. Furthermore, the function and variable that corresponds to this symbol is declared.

2. In the second pass, the μARM program is traversed and a rewrite function is generated for every function symbol that is defined. The rewrite code is deduced from the definition in the μARM program.

These passes are designed to process a μARM program which contains an arbitrary number of function definitions. In an incremental scheme, typically a small number of functions needs to be processed but sometimes complete specifications need to be translated as well. The passes can be called separately but there is also a mechanism to perform them at once. The mapping that is used is presented in Table 3.2 on page 18.

The μARM `match` statements has three arguments: a function symbol, and two tries. The first trie is executed when the outermost function symbol (OFS) of the term on top of the argument stack matches with the given function symbol, the second trie is executed when the OFS does not match with the given function symbol. Matching on multiple function symbols is achieved by a recursive structure where every second trie is a `match`. Only the innermost case has no `match` which denotes that it is the default case. An example is given in Figure 3.2. The corresponding VA code can be found in Figure 3.3.

```
dock(
   match(
      a,
      actions for a
      match(
         b,
         actions for b
         match(
            c,
            actions for c

            the default actions ))))
```

Figure 3.2 Example of μARM matching code.

```
SWITCH

    CASE( 1, a, 2 )
        actions for a

    CASE( 2, b, 3 )
        actions for b

    CASE( 3, c, 4 )
        actions for c
    DEFAULT( 4 )
        the default actions
```

Figure 3.3 Example of corresponding VA code.

The translation is implemented as an ASF+SDF specification which can be found in Appendix G on page 77. Using the ASF+SDF Meta-environment, the front end and the virtual assembler, it can be used to make a stand-alone executable version of itself (bootstrapping).

3.3 The Virtual Assembler

The code generator for VA is named *virtual assembler*. It converts VA code into actual machine code for a given architecture or into C code which is used as a portable and generic assembly language.

One of the major goals is to keep the code generator as simple as possible since it is the part that must be implemented for every target architecture. This goal is achieved by defining a *very* simple virtual assembly language that contains enough redundant information to allow the use of a naive code generator. We propose the use of a so called *macro processor* as a naive code generator. A macro processor is a program which takes a set of "rules" that each consist of a pattern and a token-string. It replaces each subsequent instance of the pattern in the input by the corresponding token-string. This process is called macro expansion. Most macro processors allow the patterns to have arguments. During expansion, these arguments are placed in the token-string. The use

μARM *instruction*	⟼	VA *instructions*

start(name)	⟼	`DEFINE( FunSym, Function, FunVar, Id )` *and*
		`FUNCTION( f_name )`
		`:`
		`END`

recycle	⟼	`RECYCLE`
goto(...)	⟼	`GOTO( ... )`
dock	⟼	`SWITCH`

match(FunSym)	⟼	`CASE(Funsym) ...`
		`:`
		`DEFAULT ...`

pairskip	⟼	`PAIRSKIP`
adrop(n)	⟼	`DROP( AP, n )`
tdrop(n)	⟼	`DROP( TP, n )`
cpshv(n)	⟼	`MV( NTH( TP, n ), PUSH( CP ) )`
apshv(n)	⟼	`MV( NTH( TP, n ), PUSH( AP ) )`
skip	⟼	`MV( POP( AP ), PUSH( TP ) )`
retract	⟼	`MV( POP( TP ), PUSH( AP ) )`
c2a	⟼	`MV( POP( CP ), PUSH( AP ) )`
ccbuild(FunName, 0)	⟼	`BUILD0( FunName, CP )`
ccbuild(FunName, 1)	⟼	`BUILD1( FunName, CP, CP )`
ccbuild(FunName, 2)	⟼	`BUILD21( FunName, CP, CP )`
acbuild(FunName, 0)	⟼	`BUILD0( FunName, AP )`
acbuild(FunName, 1)	⟼	`BUILD1( FunName, CP, AP )`
acbuild(FunName, 2)	⟼	`BUILD21( FunName, CP, AP )`
atbuild(FunName, 0)	⟼	`BUILD0( FunName, AP )`
atbuild(FunName, 1)	⟼	`BUILD1( FunName, TP, AP )`
aabuild(FunName, 2)	⟼	`BUILD21( FunName, TP, AP )`
aabuild(FunName, 0)	⟼	`BUILD0( FunName, AP )`
aabuild(FunName, 1)	⟼	`BUILD1( FunName, AP, AP )`
aabuild(FunName, 2)	⟼	`BUILD12( FunName, AP, AP )`

Table 3.2: Mapping from μARM to VA.

of a macro processor as a naive code generator is very convenient since:

- Macro processors are a standard component of all C compilers so they are available on all target architectures.
- The macro definitions can be seen as a collection of rules in a *definition file*. This is an ordinary ASCII text file which can be used on all target architectures. Consequently we have a *portable* code generator.
- The translation from VA to actual machine code (i.e., all target machine specific knowledge) is described in the macro definitions. Only these definitions need to be changed to generate code for another architecture. Consequently the code generator is easy *retargetable* (i.e., the target architecture can easily be changed).

In our scheme, we use the ANSI C preprocessor `acpp` since it is a standard preprocessor which is implemented on most architectures. Furthermore it has several features which we require, such as the ability to concatenate arguments to tokens in the token-string.

The assembler or C code generated by the virtual assembler depends completely on the rules in the definitions file. Accordingly, the user can make his own definitions to generate code that can be translated with a particular assembler or C compiler.

3.4 Translation to object code and execution

The final translation to object code is done using a native assembler or a C compiler depending on the definitions that are used. The result of this translation are chunks of object code each representing a term rewriting function. This object code can by used in two ways:

1. With the run-time system.
2. With the hybrid interpreter.

The run-time system is a skeleton which implements the environment for term rewriting functions (i.e., it implements the machine model for the abstract rewriting machine). The hybrid interpreter is an interpreter that can handle external (pre-compiled) function definitions in combination with μARM function definitions that are interpreted (see Section B.4 on page 54 for a discussion of the hybrid interpreter). The run-time system can be thought of as a hybrid interpreter that only uses external pre-compiled function definitions and does **not** interpret any μARM code.

Currently, the object code for the term rewriting functions needs to be linked with the object code of either the run-time system or the hybrid interpreter to get an executable version. Dynamic linkage of this code to the interpreter is foreseen for future versions. It is not used here since there are no ready-to-use portable tools to achieve it and the design of our own dynamic linkage tool lies beyond the scope of this project.

Chapter 4

Measurements

4.1 The benchmarks

The measurements described in this chapter were done using an ASF+SDF specification of natural numbers consisting of the constant `z` for zero and the functions `s(`N`)` for the successor of N, `plus(`N_1`,`N_2`)` for the addition of two numbers, `times(`N_1`,`N_2`)` for the multiplication of two numbers and `fac(`N`)` to denote the factorial of N (also denoted as $N!$).

This specification is translated into C code using the front end and the code generator described in this thesis. The specification and the translation are discussed in Appendix I on page 93.

The generated C code can be compiled into object code and linked to the run-time system to create a stand-alone term rewriting system (TRS). The code contains special macros for accessing the stacks, building nodes and other μARM related operations. These macros can be expanded either into (portable) ANSI C code or into GNU C code which is optimized for the Silicon Graphics architecture. Accordingly, we can make two stand-alone TRSs, one generated from portable C code and one generated from machine-specific C code. The hybrid interpreter is implemented using the same macros, so a portable and a SGI specific version of the hybrid interpreter can be made also.

Furthermore, the specification is executed with the ASF+SDF Meta-environment and it was compiled to an executable TRS using the ASF2C compiler[1]

4.1.1 Timing functions

The natural numbers specification is extended with two functions that are only used for timing purposes. The first is the `dummy(`N`)` function which replaces a arbitrary natural number with the the constant zero. This function is used to prevent large outputs from being printed (which pollutes the timing information);clearly we are not interested in the value of the computation but only in the time it takes to get to that value. `Dummy` is

[1]ASF2C [KW93] is a prototype of a compiler for ASF+SDF specifications. The complete specification is translated into one large C function which does not provide garbage collection. Consequently, ASF2C is only suitable for the compilation of specifications requiring less than, say, 60 Mb of virtual memory space. Typically these are small to medium sized applications; processing in the order of 3000 lines of input. Another drawback is the large size of the generated C code which may take a very long time to compile or cannot be compiled at all.

used in the measurements described in Section 4.2.1 to determine the average number of rewrite steps per second.

The second auxiliary function is the `reduce` function. This function repeatedly decreases a natural number with one (by peeling of an `s(...)`) until that number is equal to zero. The `reduce` function is used in the measurements described in Section 4.2.2 to determine the maximum number of rewrite steps per second.

The following should be added to the ASF+SDF specification for these functions:

```
context-free syntax
   dummy( NUM )                -> NUM
   reduce( NUM )               -> NUM
equations
   [N7]  dummy( Num )           = z
   [N8]  reduce( s( Num ) )     = reduce( Num )
   [N9]  reduce( z )            = z
```

4.2 Measurements

The machine used for the measurements is a Silicon Graphics (IRIS Indigo R4000/50 MHz with 32 MB internal memory and 8 Kb data and instruction cache) running System V (IRIX Release 4.0.5F). The reduction time is the sum of the time spent in execution of the program and the time spent in the system (i.e., the user and the system time) which were measured using the UNIX `time` command. All measurements were performed twelve times; the highest and lowest value were omitted in the computation of the average which is presented as the result here. The input term was made large enough so that running times show variation only in the last digit.

4.2.1 Average number of rewrite steps per second

The task was to compute the value of 7! for 50 times and add them together. Each time the result of 7! was replaced by zero using the dummy function. According to the debugging information of the ASF+SDF Meta-environment this task is done in 299299 rewrite steps. In Table 4.1 we present the average number of rewrite steps per second. It is computed by dividing the total number of rewrite steps (299299) by the reduction time.

	reduction time	avg. rewrite steps/sec.
ASF+SDF	54.440 sec.	5 497.78
ASF2C	1.251 sec.	239 247.80
portable μARM interpreter	3.098 sec.	96 610.39
SGI-specific μARM interpreter	1.220 sec.	245 327.05
portable TRS	1.236 sec.	242 151.29
SGI-specific TRS	0.690 sec.	433 766.67

Table 4.1 Average number of rewrite steps per second.

4.2.2 Maximum number of rewrite steps per second

The maximum number of rewrite steps can be computed by adding a very simple rewrite rule to an TRS and measure the difference in reduction time of this TRS and the TRS without this rule.

The task was to compute the value of 7! for 50 times and add them together. Each time the result of 7! was reduced to zero using the `reduce` function. According to the debugging information of the ASF+SDF Meta-environment this task is done in 551299 rewrite steps.

	reduction time
ASF+SDF	92.586 sec.
ASF2C	1.762 sec.
portable interpreter	3.780 sec.
SGI-specific interpreter	1.859 sec.
portable TRS	1.752 sec.
SGI-specific TRS	0.945 sec.

Table 4.2 Reduction time using the `reduce` function.

The `reduce` function is executed $50 * (7! + 1) = 252050$ times (once for every `s(...)` and once for every `z`). The `dummy` function in the previous measurement was executed 50 times. We can compute how long it takes to perform 252000 simple reductions by subtracting the previously measured reduction times from the times measured here. The maximum number of rewrite steps per second can be computed by dividing 252000 by this time.

	difference in reduction time	max. rewrite steps/sec.
ASF+SDF	38.146 sec.	6 606.20
ASF2C	0.511 sec.	493 150.68
portable interpreter	0.682 sec.	369 501.47
SGI-specific interpreter	0.639 sec.	394 366.20
portable TRS	0.516 sec.	488 372.09
SGI-specific TRS	0.255 sec.	988 235.29

Table 4.3 Maximum number of rewrite steps per second.

4.3 Discussion

Given the fact that these measurements are based on generated C code and not on machine specific code and that we have not implemented any optimizations (such as tail recursion elimination, partial evaluation, common subexpression elimination, or even the prevention of unnecessary movements to and from the stack) we feel that the results are satisfying and very promising.

Chapter 5

Conclusions and Future Research

5.1 Conclusions

The scheme presented in this thesis has several advantages over the direct translation from μARM to machine-specific code:

- All machine-specific steps are done using tools that are, in general, publicly available for a given target architecture.

- All machine-specific knowledge is stored in a relatively small set of definitions which typically take about 50 lines of code.

Consequently, the scheme is *easily retargetable* since a comparatively small set of definitions need to be changed to retarget the compiler to another architecture. Furthermore, the scheme is *easily portable* since the same set of definitions can be used on all architectures (the only dependency is the availability of the `acpp` preprocessor).

Furthermore, the virtual assembler also has some advantages that are essential in the context of large scale software engineering:

- it allows incremental compilation. It re-uses stable parts and only recompiles those parts that have been changed by the user or the front end. This is essential since it significantly reduces the time between making a change in the specification and being able to use the newly compiled code (the *turn around time*).

- the generated code can be used in combination with the hybrid interpreter. This allows one to compile the parts that are stable and to interpret the parts still under development. As a result the flexibility of an interpreted system is combined with the high execution speed of a compiled system.

The measurements indicate that a significant speed up in execution time can be achieved with our approach, even when C code is generated instead of machine-specific assembly language. Consequently, we feel that the results are satisfying and very promising.

5.2 Future Research

It would be very convenient if the hybrid interpreter and the run-time system have a mechanism which allows dynamic linkage of the external functions. This mechanism is not added since there is no portable approach available to do this.

Depending on the arity of the function symbol, different code has to be generated for the `match` statement. μARM does not contain enough information to let a naive code generator decide which code has to be generated. For the time being, this decision is made "dynamically" by generation an expression that is evaluated at run-time. This is an suboptimal solution since this expression must to be evaluated for all match statements. The arity of the function symbol is known in the front end (which generates the μARM code), so it is relatively easy to add it as a parameter to the `match` statement (similar to the `..build` statements). The value of this parameter can be used to decide which code must be generated.

Another optimization one could think of is combining several functions into one function to decrease the function calling overhead. When this is done naively (put all functions together) this leads to huge functions with a high turn around time and incrementality is lost. A less naive approach requires local and global analysis of the code. This introduces a lot of new aspects that have to be examined (such as call graph analysis).

The translation is currently implemented as an ASF+SDF specification. Using the ASF+SDF Meta-environment, the front end and the virtual assembler, it can be used to make a stand-alone executable version of itself (bootstrapping).

Although the generation of C code gives promising results, retargeting of the back end and generation of machine specific assembly code might be worth investigating. An imaginable approach for finding the optimal translation from VA instructions to machine specific assembly code is the use of a so called *super optimizer* [GK92]. A super optimizer is an instruction sequence generator that uses a exhaustive generate-and-test approach to find the shortest instruction sequence for a certain function on a given architecture. After specifying the function, the target architecture, and the maximum sequence length, the program generates all instruction sequences that can be used to implement that function and selects the shortest one. The time complexity of the used algorithm is reported to be $O\ (m\ *\ n^2)^n$ where m is the number of available instructions on the architecture and n is the shortest sequence for the goal function. Consequently, the use of a super optimizer takes a lot of time but since it is used only once, this effort is expected to be counterbalanced by an acceptable increase in execution speed of the generated assembly code.

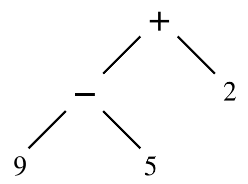
Glossary

Terms and phrases that are explained elsewhere in the glossary are *emphasized*.

A

abstract rewriting machine: an imaginary machine with three stacks, a heap and several internal registers. The machine performs an iterative rewriting process in which it is guided by a μARM program. The rewriting process repeatedly attempts to recognize an instantiation of the left-hand side of a rule (a redex). Then, possibly after checking conditions (which involves a recursive invocation of the rewriting process), the established redex is replaced by an instantiation of the right-hand side of that rule. The guiding program typically consists of instructions for the matching of function symbols, the creation of terms on the heap and operations on the stacks.

abstract syntax tree: a tree in which each node represents an operator and the children of that node represent the operands of that operator.



The abstract syntax tree for $9 - 5 + 2$.

Argument Stack (AS): One of the three μARM stacks. It contains references to trees which have been normalized, and for which a parent function symbol still has to be processed.

ARM: the first generation of the abstract rewriting machine. This is the two stack machine that is described in [KW93]. (see also *abstract rewriting machine* and *μARM*).

AS: abbreviation for *Argument Stack*.

ASF+SDF: a modular specification formalism for the integrated definition of syntax and semantics which is the result of the combination of the two formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism).

assembly language: the mnemonic code which is translated into object code using an assembler. In this thesis, the phrase "assembly language" is used interchangeably with the phrase "machine language".

B

back end: (also called code generator) the part of the compiler that translates the intermediate representation created by the front end into machine code. It is the part of the compiler that is concerned with all architecture dependent decisions.

C

CGD: abbreviation for *Code Generator Description*.

CGG: abbreviation for *Code Generator Generator*.

code generator: (also called back end) the part of the compiler that translates the intermediate representation created by the front end into machine code. It is the part of the compiler that is concerned with architecture dependent decisions.

Code Generator Description (CGD): a "database" holding machine language commands that correspond to constructs of an input language. The CGD is used by a code generator generator to builds a tree pattern matcher which can be used to find the optimal machine language code for a given input program.

Code Generator Generator (CGG): a system that automatically builds a code generator (or back end) out of a relatively short declarative description.

conditional equation: an equation that has one or more conditions that need to be satisfied. The conditions are evaluated and the equation is only used when all conditions are valid.

context-free syntax: The context-free syntax describes the concrete and abstract syntactic structure of sentences in a language. In ASF+SDF a context-free syntax contains a set of declarations for so called context-free functions, each consisting of zero or more literals, sort names or lists, and followed by '`->`' and the result sort. These functions may be followed by attributes that control how parentheses and brackets affect the abstract syntax or by attributes that define the associativity of a rule. All functions with the same result sort together define the alternatives for that sort.

Control Stack (CS): One of the three μARM stacks. It serves two purposes. First, it contains all function symbols still to be evaluated. Secondly, it can hold certain arguments to represent an intermediate stage of the rewriting process.

cover: A cover is a collection of patterns that completely describes the input.

CS: abbreviation for *Control Stack*.

CVM: abbreviation for C Virtual Machine. This is the virtual (or abstract) machine that is used as an intermediate stage in the `VPCC` compiler.

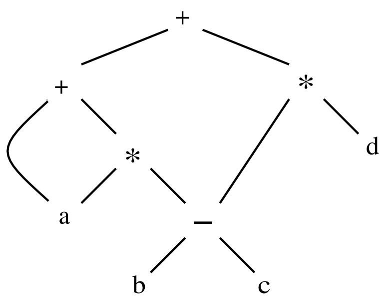
D

dag: abbreviation for *directed acyclic graph*.

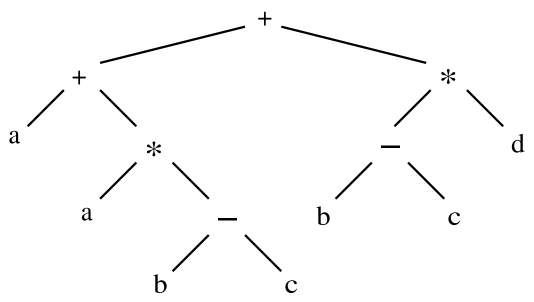
definition file: The file that contains the rules for the, architecture dependent, mapping from VA to actual machine code. It is used by the *virtual assembler*.

degree of incrementality: (or *grain size*) a measurement for the size of the smallest change that is passed through to the next stage of an incremental compiler.

directed acyclic graph (dag): a graph that has directed edges and contains no cycles (i.e., it is not possible to return in a node by following the edges). Directed acyclic graphs are used to combine common subexpressions in an expression. Like an *abstract syntax tree*, a dag has a node for every subexpression; an interior node represents an operator and its children represent its operands. The difference between a dag and an abstract syntax tree is that a node in a dag which represents a common subexpression has more than one parent. In the corresponding abstract syntax tree, the common subexpression would be presented as a duplicated subtree.



The directed acyclic graph for $a + a * (b - c) + (b - c) * d$.



The abstract syntax tree for $a + a * (b - c) + (b - c) * d$.

E

EDB: abbreviation for Equation Debugger, the debugging facility of the ASF+SDF Meta-environment.

external functions: functions that are used but not defined in an ASF+SDF specification. These functions receive their definition by linking the external function code to the interpreter.

expression tree: a tree representation of an expression, for example, an abstract syntax tree. Expression trees are frequently used as an intermediate representation between the front end and the back end of a compiler.

F

front end: the part of the compiler that translates source code into an intermediate representation. It is the language dependent part of a compiler and usually does all kinds of global optimizations etc.

G

GCC: abbreviation for Gnu C Compiler [Sta89].

GCG: abbreviation for Generated Code Generator.

grain size: size of the smallest change that is passed through to the next stage of an incremental compiler (see also *degree of incrementality*).

granularity: see *degree of incrementality*.

H

hybrid interpreter: an interpreter that can handle external (pre-compiled) function definitions in combination with function definitions in a μARM program that is interpreted.

host architecture: the architecture on which the process is executed.

I

incrementality: the ability of a compiler to reuse stable parts and only recompile those parts that have been changed.

Intermediate Representation (IR): a representation (e.g. *abstract syntax trees* or a language) that is used to transfer data between two parts of a program (e.g., the transfer of a partially processed program between the front end and the back end of a compiler).

IR: abbreviation for *Intermediate Representation*.

L

lexical syntax: The lexical syntax describes the low level structure of text by means of *lexical tokens*. A lexical token consists of a sort name (used to distinguish classes of tokens like identifiers and numbers), and the actual text of the token. The lexical syntax also defines which substrings of the text are layout symbols or comments and are to be skipped.

A lexical syntax contains a set of declarations for lexical functions, each consisting of a regular expression and a result sort separated by '`->`'. All functions with the same result sort together define the lexical syntax of tokens of that sort. Regular expressions may contain strings, sort names, character classes (i.e., enumerations of characters enclosed by '[' and ']'), and repetition operators ('*' and '+'). Spaces are only significant inside strings and character classes. Circular dependencies between declarations in the lexical syntax are forbidden.

lexical token: Lexical tokens are the elementary particles of a language. They are used to define the *lexical syntax* of a language. In ASF+SDF, lexical tokens consist of a sort name (used to distinguish classes of tokens like identifiers and numbers), and the actual text of the token.

M

μARM: the current generation of the abstract rewriting machine (see also *abstract rewriting machine* and *ARM*).

machine language: the mnemonic code which is translated into object code using an assembler. In this thesis, the phrase "machine language" is used interchangeably with the phrase "assembly language".

macro processor: a program which takes a set of "rules" that each consist of a pattern and a token-string. The macro processor replaces each subsequent instance of the pattern in the input by the corresponding token-string. This is called macro expansion. Most macro processors allow the patterns to have arguments. During the expansion, these arguments are placed in the token-string.

minimal cover: A minimal cover is a collection of patterns (each having a cost factor) that describes the complete input with minimal costs.

N

naive code generation: code generation where code for the intermediate instructions is emitted without being optimized (for example, statically known variables are **not** replaced by constants).

naive code generator: a code generator that emits code for a given (intermediate) instruction without optimizing the generated code (for example, statically known variables are **not** replaced by constants).

native assembler: an assembler that is designed for and implemented on a specific target architecture.

normal form: a term that cannot be further reduced using a given term rewriting system.

O

OFS: abbreviation for outermost function symbol

P

portable: the ability to (easily) change the host architecture (i.e., the architecture on which the process is executed).

R

redex: abbreviation for reducible expression. This is an instantiation of the left-hand side of a rewrite rule.

register allocator: the process in a *code generator generator* that is concerned with the determination of which registers are to be used by the emitted machine instructions.

Register Transfer Language (RTL): [MRS90, JM91] an intermediate representation; it is code for a (virtual) machine with an infinite number of registers. In RTL, the machine instructions to be generated are described in an algebraic form which describes the *semantics* (what the instruction does). RTL is used for machine-independent optimization.

retargetable: the ability to (easily) change the target architecture (i.e., the architecture for which machine code is generated).

rewriting: see *term rewriting*.

RTL: abbreviation for *Register Transfer Language*.

run-time system: a skeleton which implements the environment for term rewriting functions (i.e., it implements the machine model of the abstract rewriting machine). The run-time system is linked together with the term rewriting functions to build a stand-alone system.

S

semantics: the meaning (of an instruction, function, program, etc.).

signature: definition of the abstract syntax of the functions used in an SDF specification.

super optimizer: an instruction sequence generator that uses a exhaustive generate-and-test approach to find the shortest instruction sequence for a certain function on a given architecture. After specifying the function, the target architecture, and the maximum sequence length, the program generates all instruction sequences that can be used to implement that function and selects the shortest one.

syntax: the way a language, instruction, statement, etc. looks like.

T

target architecture: the architecture for which machine code is generated.

term rewriting: an iterative process which repeatedly attempts to recognize an instantiation of the left-hand side of a rule (a redex). Then, possibly after checking conditions (which involves a recursive invocation of the rewriting process), the established redex is replaced by the corresponding instantiation of the right-hand side of that rule.

term rewriting system (TRS): a finite automate which uses a collection of *rewrite rules* to perform a term rewriting process (see *term rewriting*). The rewrite rules consist of a left hand side, a right hand side and, possibly, some conditions.

Traversal Stack (TS): One of the three μARM stacks. It contains, during matching, all nodes that have been visited. This allows for easy backtracking since previous states can be retrieved. The traversal stack is used only during matching; it has no meaning in subsequent rewrite steps.

tree pattern matcher: a program which analyses a tree and performs actions (e.g., it generates code) when a certain pattern is discovered.

trie In the context of μARM, a trie (pronounced as try) is a (part of a) μARM program. Trie is the abbreviation for Term Recognizing and Instantiating Entity. It contains one or more μARM statements and is typically used to denote a composition of instructions that can be treated as one instruction (for example the instructions that have to be executed when a match succeeds).

TRS: abbreviation for *term rewriting system.*

TS: abbreviation for *Traversal Stack.*

turn around time: the time between making a change in the specification and being able to use the newly compiled code.

V

VA: abbreviation for *virtual assembly language.*

virtual assembly language: the intermediate language that is used in the back end of the ASF+SDF compiler.

virtual assembler: the code generator of the ASF+SDF compiler. It reads a program in the virtual assembly language and emits machine code or C code depending on the *definitions* that are used.

VPCC: abbreviation for *Very Portable C Compiler.*

Bibliography

[AM87] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 301–24, New York, 1987. Springer-Verlag.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley, 1986.

[AU73] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling - Volume II: Compiling*. Series in Automatic Computation. Prentice-Hall, 1973.

[BDB90] A. Balachandran, D. M. Dhamdhere, and S. Biswas. Efficient retargetable code generation using bottom-up tree pattern matching. *Computer Languages*, 15(3):127–140, 1990.

[Ber87] J.A. Bergstra. *Terminologie van algebraische specificaties*. Kluwer Programmatuurkunde, 1987. In Dutch.

[BHK87] J.A. Bergstra, J. Heering, and P. Klint. ASF - an algebraic specification formalism. Report CS-R8705, Centrum voor Wiskunde en Informatica (CWI), Amsterdam, 1987.

[BHK89] J.A. Bergstra, J. Heering, and P. Klint, editors. *Algebraic Specification*. ACM Press Frontier Series. The ACM Press in co-operation with Addison-Wesley, 1989.

[BKM89] J.A. Bergstra, J.W. Klop, and A. Middeldorp. *Termherschrijfsystemen*. Kluwer Bedrijfswetenschappen, 1989. In Dutch.

[Bou91] W. Bouma. *Algebraische Specificaties*. Kluwer Bedrijfswetenschappen, 1991. In Dutch.

[Emm90] H. Emmelmann. *BEG - a Back End Generator - User Manual*. GMD Forschungsstelle an der Universität Karlsruhe, April 1990. Version 1.

[ESL89] H. Emmelmann, F-W. Schröer, and R. Landwehr. BEG - a Generator for Efficient Back Ends. In *Proceedings of the Sigplan '89 Conference on*

Programming Language Design and Implementation. ACM Press, July 1989. Appears as SIGPLAN Notices, Vol. 24, Number 7.

[FH91a] C.W. Fraser and D.R. Hanson. A code generation interface for ANSI C. *Software – Practice & Experience*, 21(9):963–988, September 1991.

[FH91b] C.W. Fraser and D.R. Hanson. A retargetable compiler for ANSI C. Technical Report CS-TR-303-91, Department of Computer Science, Princeton University, February 1991.

[FH91c] C.W. Fraser and D.R. Hanson. A retargetable compiler for ANSI C. *SIGPLAN Notices*, 26(10):29–43, October 1991.

[FHP92a] C.W. Fraser, D.R. Hanson, and T.A. Proebsting. Engineering a simple, efficient code generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992. iburg is available by anonymous ftp from ftp.cs.princeton.edu, file /pub/iburg.tar.Z.

[FHP92b] C.W. Fraser, R.R. Henry, and T.A. Proebsting. BURG - fast optimal instruction selection and tree parsing. *SIGPLAN Notices*, 27(4):68–76, April 1992. BURG is available by anonymous ftp from kaese.cs.wisc.edu, file /pub/burg.shar.Z.

[GK92] Torbjörn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the GNU CC compiler. In *Proceedings of the Sigplan '92 Conference on Programming Language Design and Implementation*. ACM Press, July 1992. Appears as SIGPLAN Notices, Vol. 27, Number 7.

[HHKR89] J. Heering, P.R.H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF - reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, Edinburgh, 1986.

[HU79] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[JM91] Ralph E. Johnson and Carl D. McConnell. The RTL system: A framework for code optimization. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991. Available by anonymous ftp from speedy.cs.uiuc.edu, file /pub/papers/RTL-System.ps.

[Joh86] S.C. Johnson. *YACC: yet another compiler-compiler*. Bell Laboratories, 1986. UNIX Programmer's Supplementary Documents, Volume 1 (PS1).

[Kli93] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[Koo92] J.W.C. Koorn. GSE: A generic text and structure editor. Report P9202, University of Amsterdam, 1992.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language.* Prentice Hall, second edition, 1988.

[KW93] J. F. Th. Kamperman and H.R. Walters. ARM, abstract rewriting machine. Technical Report CS-9330, Centrum voor Wiskunde en Informatica, 1993. Available by *ftp* from ftp.cwi.nl:/pub/gipe as KW93.ps.Z.

[McC92] Carl D. McConnell. *Tree-Based Code Optimization.* Thesis proposal, University of Illinois at Urbana-Champaign, May 1992. Available by anonymous ftp from speedy.cs.uiuc.edu, file /pub/papers/tree-based-opt.prelim.ps.

[McC93] Carl D. McConnell. *Tree-Based Code Optimization.* Abandoned thesis, University of Illinois at Urbana-Champaign, May 1993. Available by anonymous ftp from speedy.cs.uiuc.edu, file /pub/papers/tree-based-opt.ps.

[MRS90] Carl D. McConnell, J. David Roberts, and Charles B. Schoenig. The RTL system. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, October 1990. Available by anonymous ftp from speedy.cs.uiuc.edu, file /pub/papers/detailed-RTL-System.ps.

[Rek92] J. Rekers. *Parser Generation for Interactive Environments.* PhD thesis, University of Amsterdam, 1992. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Rek92.ps.Z.

[Sta89] R.M. Stallman. *Using and Porting GNU CC.* Free Software Foundation, second edition, September 1989.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of computer Science, Carnegie Mellon University, November 1990.

[TvSKS83a] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson. Description of a machine architecture for use with block structured languages. Technical Report IR-81, Vrije Universiteit Amsterdam, August 1983.

[TvSKS83b] A.S. Tanenbaum, H. van Staveren, E.G. Keizer, and J.W. Stevenson. A practical toolkit for making portable compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[Wal91] H.R. Walters. *On Equal Terms, Implementing Algebraic Specifications.* PhD thesis, University of Amsterdam, 1991. Available by *ftp* from ftp.cwi.nl:/pub/gipe as Wal91.ps.Z.

Appendix A

ASF+SDF

A.1 The ASF+SDF specification formalism

The ASF+SDF formalism [BHK89] is the combination of the two formalisms ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism). The ASF formalism [BHK87] is based on the notion of a module consisting of a *signature* defining the abstract syntax of functions and a set of *conditional equations* defining their semantics. The SDF formalism [HHKR89] allows the simultaneous definition of concrete (i.e., lexical and context-free) and abstract syntax and implicitly defines a translation from text strings to abstract syntax trees.

The main idea of ASF+SDF is to identify the abstract syntax defined by the signature in ASF specifications with the abstract syntax defined implicitly by an SDF specification. This results in a standard mapping from text to abstract syntax tree which gives the possibility to associate semantics with the tree representation of text and to introduce user-defined notation in specifications. ASF+SDF is therefore a modular specification formalism for the integrated definition of syntax and semantics.

A.1.1 Modules and Modular Structure

An ASF+SDF specification consists of a sequence of module declarations. Each module may define syntax rules as well as semantic rules and the notation used in the semantic rules depends on the definition of syntax rules. The entities declared in a module may be visible or invisible to other modules. A module can use another module from the specification by importing it. As a result, all visible names of the imported module become available in the importing module. The overall structure of a module is:

imports
    Names-of-imported-modules
*Section**
priorities
    Priority-relations-between-context-free-rules
equations
    Conditional-equations

A module thus consists of an imports clause followed by a list of zero or more sections (that define sorts, lexical syntax, context-free syntax and variables) optionally followed by a priority declaration and conditional equations.

Each *Section* starts with one of the keywords `exports` or `hiddens`; `exports` makes all entities in the section visible to other modules, `hiddens` makes all entities in the section local to the module. A *Section* has the form:

exports-or-hiddens
    `sorts`
        Names-of-sorts
    `lexical syntax`
        Rules-of-the-lexical-syntax
    `context-free-syntax`
        Rules-of-the-context-free-syntax
    `variables`
        Names-of-variables

where each of the subsections for sorts, lexical syntax, context-free syntax and variables are optional.

Modules can be combined by importing one module in another. As a result all exported entities of the imported module (and of all modules that are imported indirectly by it) become available in the importing module.

A.1.2 Sorts

A sort is declared by listing its name in a `sorts` section of a module. A sort name should start with a capital letter and may be followed by letters and/or digits. Hyphens ('`-`') and underscores ('`_`') may be embedded in sort names. There are four predefined sorts (`LAYOUT, IGNORE, REJECT, CHAR`) which may *not* be redeclared.

A.1.3 Lexical Syntax

The lexical syntax describes the low level structure of text by means of *lexical tokens*. A lexical token consists of a sort name (used to distinguish classes of tokens like identifiers and numbers), and the actual text of the token. The lexical syntax also defines which substrings of the text are layout symbols or comments and are to be skipped.

A lexical syntax contains a set of declarations for *lexical functions*, each consisting of a regular expression and a result sort separated by '`->`'. All functions with the same result sort together define the lexical syntax of tokens of that sort. Regular expressions may contain strings, sort names, character classes (i.e., enumerations of characters enclosed by '`[`' and '`]`'), and repetition operators ('`*`' and '`+`'). Spaces are only significant inside strings and character classes. Circular dependencies between declarations in the lexical syntax are forbidden.

All literal symbols such as keywords and operator symbols that appear in the context-free syntax are added to the lexical syntax implicitly.

Lexical Ambiguities

Lexical ambiguities are resolved by the following four rules:

- *Prefer Longest Match per Sort*: reject all interpretations of the input text that are included in a longer interpretation of the same sort. Given a standard definition of identifiers, the input '`xyz`' will thus lead to recognition of the identifier '`xyz`' and *not* to either '`x`' or '`xy`'.

- *Prefer non-Layout*: eliminate all interpretations of the text as layout symbol. When, after applying this rule, no interpretations remain, the next lexical token is taken from the input text.
- *Prefer Literals*: give preference to interpretation as a literal, i.e., as keyword or operator introduced in the context-free syntax, over interpretation as a more general symbol (such as identifier).
- *Prefer variables*: give preference to interpretation as a variable (as defined in a variables section) over interpretation as a literal.

Ambiguities that cannot be solved by these rules remain, and are passed on to the syntax analysis phase.

A.1.4 Context-free Syntax

The context-free syntax describes the concrete and abstract syntactic structure of sentences in a language. A context-free syntax contains a set of declarations for *context-free functions*, each consisting of zero or more literals, sort names, or lists followed by '`->`' and a result sort. They may be followed by attributes that control how parentheses and brackets affect the abstract syntax or by attributes that define the associativity of a rule. All functions with the same result sort together define the alternatives for that sort. Consider the following language of coordinates and drawing commands:

```
sorts NAT COORD CMND
lexical syntax
   [0-9]+              -> NAT
   [ \n\               -> LAYOUT
context-free syntax
   "(" NAT "," NAT ")"  -> COORD
   "line" "to" COORD    -> CMND
   "move" "to" COORD    -> CMND
```

Written as a conventional BNF grammar (and not considering lexical syntax) this is equivalent to:

```
<COORD> ::= "(" <NAT> "," <NAT> ")"
<CMND>  ::= "line" "to" <COORD> | "move" "to" <COORD>
```

List Structures

It happens quite often that a context-free syntax requires the description of the repetition of a syntactic notion or of list structures (with or without separators) containing a syntactic notion. ASF+SDF provides the following primitives for this:

- *Lists without separators*

  S* defines zero or more repetitions of sort S.

  S+ defines one or more repetitions of sort S.

- *Lists with separators*

$\{S\ sep\}$* defines zero or more repetitions of sort S separated by the literal *sep*.

$\{S\ sep\}$+ defines one or more repetitions of sort S separated by the literal *sep*.

Lists may be used in the left-hand side of a context-free function and in the right-hand side of a variable declaration. They may **not** be used as result sort in the right-hand side of a context-free function. This imposes no real restriction since a new sort can be introduced representing a list. This new sort may then be used in the left-hand side of other context-free rules.

Using lists, the syntax of the declaration of identifiers in a Pascal-like language can be defined as follows:

```
sorts ID DECL TYPE
lexical syntax
   [a-z]+                      -> ID
   " \n\t"                     -> LAYOUT
context-free syntax
   var { ID "," }+ ":" TYPE    -> DECL
   integer                     -> TYPE
   real                        -> TYPE
```

A.1.5 Priorities

The context-free syntax defined in an ASF+SDF specification may be ambiguous: there are sentences which have more than one associated tree. The common example are the arithmetic expressions in which definitions of the priority or associativity of operators are missing. In ASF+SDF there are three mechanisms for defining associativity and priority of functions:

1. Relative priorities of functions (defined in the '`priorities`' section). This is used to define which functions bind stronger than other functions.
2. Associativity of functions (defined with one of the attributes `left`, `right`, `assoc` or `non-assoc` following the function declaration). This is used to solve ambiguities introduced by nested occurrences of binary operators.
3. Associativity of groups of functions (defined in the '`priorities`' section). This is used to solve ambiguities introduced by functions with the same priority.

A.1.6 Variables

Variables are declared in the '`variables`' section of a module. It consists of a list of variables together with their sort. Each declaration defines a *naming scheme* for variables together with a result sort. A naming scheme is a regular expression like the ones allowed in the lexical syntax except that sorts are **not** allowed. One variable declaration may thus define an unlimited number of variables. The sort of a variable may be a simple sort, or a list with or without separators. The variables '`Id`', '`Type3`', and '`Id-List''`' are examples of variables declared by the following specification:

```
sorts ID DECL TYPE
lexical syntax
```

```
      [a-z]+                         -> ID
      " \n\t"                        -> LAYOUT
   context-free syntax
      decl { ID "," }+ ":" TYPE      -> DECL
      integer                        -> TYPE
      real                           -> TYPE
   variables
      Id                             -> ID
      Type [0-9]*                    -> TYPE
      Id-list [']*                   -> { ID "," }*
      Id-ne-list                     -> { ID "," }+
```

A.1.7 Equations

With equations a *meaning* or *semantics* may be added to functions declared in the context-free syntax. Equations establish equalities between terms. The equality of two terms S and T is defined by an *unconditional equation* of the form:

[*TagId*] S = T

where *TagId* is a sequence of letters and/or digits starting with an uppercase letter or a digit, and S and T must be terms of the same sort. A *conditional equation* consists of a number of *conditions* (also called *premises*) and a conclusion. It can be written in three (syntactically different) ways:

```
(1)   [TagId]   S = T  when C1, C2, ...
(2)   [TagId]   C1, C2, ...   ====> S = T
(3)   [TagId]   C1, C2, ...
                ===========
                  S = T
```

where $C1$, $C2$, ... are conditions which may be either *positive* (and have the form 'S = T'), or *negative* (and have the form 'S != T').

Note that list sorts are **not** allowed as the sort of the term at the left-hand side or right-hand side of a condition or conclusion. This restriction can be circumvented by introducing a new sort for the list.

A.1.8 The module `Bool-example`

One of the simplest specifications possible, and therefore frequently used as an example, is the data type of the Boolean values. It defines the constants `true` and `false` and the functions *and* and *or* (written in infix notation using the left-associative operators '`&`' and '`|`', respectively) and *not* (written in prefix notation using the function symbol '`not`'). Here is the specification:

```
   module Bool-example
      exports
         sorts BOOL
         lexical syntax
            "%%" ~[\n]* [\n]        -> LAYOUT
            [ \t\n]                 -> LAYOUT
```

```
    context-free syntax
      true                    -> BOOL
      false                   -> BOOL
      BOOL "|" BOOL           -> BOOL {left}
      BOOL "&" BOOL           -> BOOL {left}
      not "(" BOOL ")"        -> BOOL
      "(" BOOL ")"            -> BOOL {bracket}
    variables
      Bool [0-9']*            -> BOOL

  priorities
    "|" < "&"

  equations

  %% disjunction
    [B1]   true | Bool  = true
    [B2]   false | Bool = Bool

  %% conjunction
    [B3]   true & Bool  = Bool
    [B4]   false & Bool = false

  %% negation
    [B5]   not(false)   = true
    [B6]   not(true)    = false
```

A.2 The Meta-environment

A.2.1 Introduction

The ASF+SDF Meta-environment [Kli93] is an interactive development environment for the automatic generating of interactive environments for manipulating programs, specifications, or other texts written in a formal language. The generation process is controlled by a definition of the target language, which typically includes features as syntax, pretty printing, type checking and execution of programs. The ASF+SDF Meta-environment can be helpful when:

- one has to write a formal specification for some problem and needs interactive support to do this.
- one has developed an application language and wants to create an interactive environment for it.

The ASF+SDF formalism allows the definition of syntactic as well as semantic issues. It can be used for the definition of languages (for programming, for writing specifications, for querying databases, for text processing, or for dedicated applications). In addition it can be used for the formal specification of a wide variety of problems. ASF+SDF provides:

- A general-purpose algebraic specification formalism based on equational logic.
- Modular structuring of specifications.
- Integrated definition of lexical, context-free and abstract syntax.
- User-defined syntax, allowing one to write specifications using their own notation.
- Complete integration of the definition of syntax and semantics.
- Hybrid functions, allowing one to define functions using Lisp instead of equations.

Furthermore, the ASF+SDF Meta-environment offers:

1. Syntax-directed editing of ASF+SDF specifications.
2. Incremental compilation and testing of specifications.
3. Compilation of ASF+SDF specifications into dedicated interactive environments containing various tools such as a lexical analyzer, a parser, a pretty printer, a syntax-directed editor, a debugger, and an interpreter or compiler.

The advantages of creating interactive environments in this way are twofold:

Increased uniformity: Similar tools for different languages often suffer from a lack of uniformity. Generating tools from language definitions will result in a large increase in uniformity, with corresponding benefits for the user.

Reduced implementation effort: Preparing a language definition requires significantly less effort than developing an environment from scratch.

A.2.2 Global Structure of the Meta-environment

Typical specifications consist of a series of modules. Individual modules can be edited by invoking a *module editor*. All editing in the Meta-environment is done by creating instances of the syntax-directed editor GSE (Generic Syntax-directed Editor) and a module editor is a special example of such an instance.

After each editing operation on a module its *implementation* is updated immediately. It consists of a lexical scanner, a parser, a pretty printer, and a term rewriting system which are all derived from the module automatically.

A module can be tested by invoking a *term editor* to create and evaluate terms defined by the module. Term editors use the syntax of the module for parsing the textual representation of terms and for converting them to internal format (abstract syntax trees). The equations of the module are then used to reduce the terms into normal form. This result is, in its turn, converted back to textual form by pretty printing it.

The reduction of a term can be monitored by using *the Equation Debugger (EDB)*, a debugging system that allows one to reduce a term step-by-step or to place breakpoints for interrupting the reduction process at specific moments.

Finally, term editors can be customized by adding *buttons* whose activation starts the evaluation of a function which is either defined in the specification or is implemented in Lisp. In this way the user-interface of the application can be customized by adding, for instance, a type-checking or evaluation button to a term editor.

A.2.3 Syntax-directed Editing

All editing in the ASF+SDF Meta-environment is done with the syntax-directed editor GSE in which textual editing and structural editing are highly integrated. GSE tries to help with writing syntactically correct texts. This is achieved by providing a "focus" on the text. All the text *outside* the focus is (by definition) always syntactically correct. The text *inside* the focus is displayed by surrounding it with a solid line and may be the result of arbitrary textual editing operations. When the focus is "moved" to another area in the text, the syntax of the text in the old focus area is first checked. If it is correct, the focus is moved to the new area. If it is not correct, a warning is displayed and the error should be corrected before proceeding.

Appendix B

The Abstract Rewriting Machine

The front end of the compiler translates an ASF+SDF specification into intermediate code which implements a term rewriting system on the *abstract rewriting machine.* This is an abstract machine that is dedicated to term rewriting. The current generation of this machine is named μARM. It is based on the Abstract Rewriting Machine described in [KW93].

In the remainder of this appendix, we will describe the rewriting process, give an overview of the μARM machine model and discuss its language.

B.1 Abstract rewriting

Rewriting is an iterative process, which repeatedly attempts to recognize an instantiation of the left-hand side of a rule (a redex). Then, possibly after checking conditions (which involves a recursive invocation of the rewriting process), the established redex is replaced by an instantiation of the right-hand side of that rule.

In accordance, the abstract rewriting machine performs a cyclic process in which it is guided by a μARM program. The programming language μARM has control structures for analyzing terms, recursively calling the rewrite process, and generating new terms. These control structures use and manipulate the machine's registers, stacks and its heap. There are no instructions to manipulate these memories explicitly, nor can arithmetic or logic computations be performed, since μARM only concerns rewriting.

The cyclic process can be described as follows. Initially all function symbols in a subject term are pushed in pre-order on a stack (called the control stack), which for that moment may be regarded as the input stack. In general, no tree structures will be made by μARM for terms which have not yet been processed. Consequently, an actual tree representation is made only if a (sub-)term is a normal form.

Each cycle a symbol is taken from the control stack and is interpreted as the outermost function symbol of a to-be-rewritten term. The immediate sub-terms of this term are found (that is, if the function symbol is not a constant) on a second stack, the argument stack. As required, the sub-terms are analyzed further, until a rewrite rule matches. Then conditions are checked and the corresponding right-hand side is substituted. This is done by pushing the appropriate symbols (and sub-terms of terms on the argument stack) onto the control stack.

If no rewrite rule matches, a normal form is constructed from the function symbol on the control stack and the required number of terms from the argument stack.

Execution of this process continues until the control stack is empty, at which time the argument stack holds the normal form of the input term. Below, we will discuss this process in more detail.

B.2 The μARM machine model

The μARM machine has three stacks and a heap. The heap contains the trees representing the (sub-)terms. A single primitive exists for the creation of tree nodes. Destruction occurs implicitly by a garbage collector which removes inaccessible data.

The first stack is called the *control stack* (CS), and it serves two purposes. First, it contains all function symbols still to be evaluated. Initially it contains precisely all function symbols in the subject tree, pushed onto it in a prefix order. Note that no tree parts have to be made for these function symbols, because they will probably be rewritten anyway.

The second stack is the *argument stack* (AS). It contains references to trees which have been normalized, and for which a parent function symbol still has to be processed (μARM has a built-in innermost reduction strategy).

The second purpose of the control stack is to hold certain arguments to represent an intermediate stage of the rewriting process. When a rule is applied, the instantiated right-hand side is entirely pushed on the control stack, in prefix order. If, at the beginning of a new cycle, μARM finds a tree reference (i.e., a normal form) on the control stack it is simply pushed onto the argument stack and the next cycle may start.

The third stack is the *traversal stack (TS)* which contains, during matching, all nodes that have been visited. The traversal stack allows for easy backtracking during the matching of an term. Each successful match on a single function symbol causes that node to be passed to the traversal stack, while its arguments are pushed on the argument stack. When matching fails at some point, the previous (correct) state can be retrieved from the traversal stack. The traversal stack is used only during matching; it has no meaning in subsequent rewrite steps.

B.2.1 Example of the use of the control- and the argument stack

As an example, we will describe μARM's normalization of:

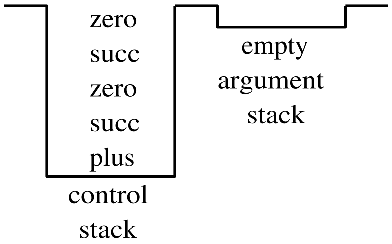
```
plus(succ(zero),succ(zero))
```

in the term rewriting system (TRS) consisting of the rules:

```
plus(zero,Nat) = Nat
plus(succ(Nat1),Nat2) = succ(plus(Nat1,Nat2))
```

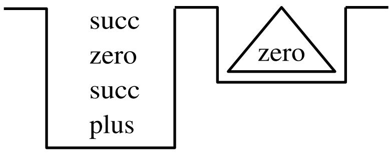
In this system `zero` is a constant, `succ` and `plus` are functions and `Nat`, `Nat1` and `Nat2` are variables.

In the pictures below only the control stack and the argument stack are shown since the traversal stack has no meaning in subsequent rewrite steps. An example describing the use of the traversal stack will follow.

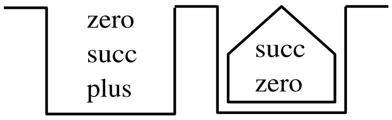


Initially, the control stack contains all function symbols in the subject term.

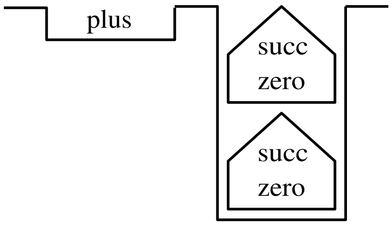
Step 1. No rule for `zero` exists, so the default action is taken: a tree node is built, and a reference to that node (here symbolized by a triangle) is put on the argument stack.



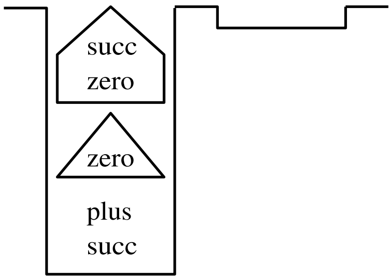
Step 2. Again no rules exist. Since `succ` is a unary function, a single argument is taken off the argument stack, and combined with `succ` into the tree `succ(zero)`. A reference to this tree is put back on the argument stack.



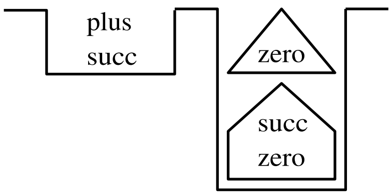
Step 3 & 4. Again, no rules exist for `zero` and `succ`, and the following situation results.



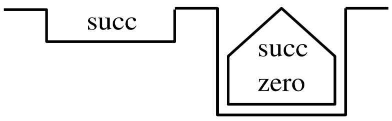
Step 5. Now, the control stack has `plus` on top, and the outermost symbol of the first item on the argument stack is `succ`. Hence rule 2 applies. After removing the arguments of `plus`, its right-hand side is traversed in pre-order, and is pushed completely on the control stack.



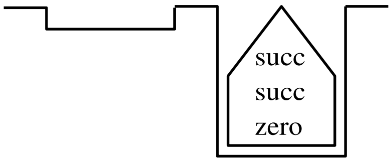
Step 6. Observe the distinction between, e.g., `plus`, which is a function symbol, and the *tree* that contains `zero`. Note that the trees that were pushed on the control stack, will immediately be moved back to the argument stack. An (unimplemented) optimization is to push these trees directly onto the argument stack these trees.



Step 7. Plus with `zero` as first argument results in the second argument being pushed on the argument stack.



Step 8. Succ combined with `succ(zero)` on top of the argument stack term results in the tree for `succ(succ(zero))` being pushed on the argument stack.



Step 9. As we can see, the input term is normalized as soon as the control stack is empty. At this time, the argument stack holds exactly one (reference to a) term. This is the required normal form.

B.2.2 Example of the use of the traversal stack

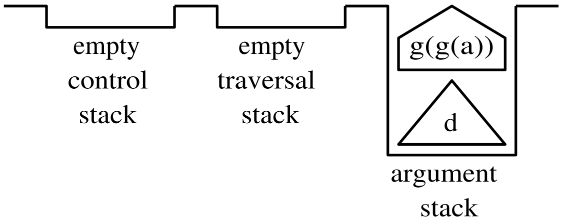
This example illustrates the use of the traversal stack. It describes the last stage of the normalization of `f(g(g(a)),d)` in the TRS consisting of the rules:

```
f(g(g(b)),c) = a
f(g(X),d) = b
```

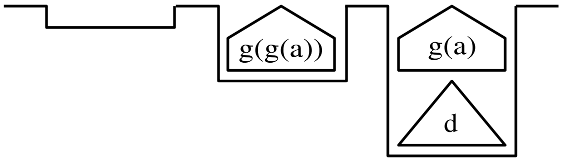
In this system `a`, `b`, `c` and `d` are constants, `f` and `g` are functions and `X` is a variable. The following μARM code is generated for the rewriting of `f`:

```
start(
   f,
   dock(
      match(
         g,
         dock(
            match(
               g,
               dock(
                  match(
                     b,
                     dock(
                        match(
                           c,
                           acbuild(a,0,tdrop(4,recycle)),
                           tdrop(1,
                              dock(
                                 match(
                                    d,
                                    acbuild(b,0,tdrop(3,recycle)),
                                    tdrop(1,skip(atbuild(f,2,recycle))))))),
                     adrop(1,
                        dock(
                           match(
                              d,
                              acbuild(b,0,tdrop(3,recycle)),
                              tdrop(1,skip(atbuild(f,2,recycle))))))),
               skip(
                  dock(
                     match(
                        d,
                        acbuild(b,0,tdrop(3,recycle)),
                        tdrop(1,skip(atbuild(f,2,recycle)))))))),
         skip(skip(atbuild(f,2,recycle))))),
  notrie))))))
```

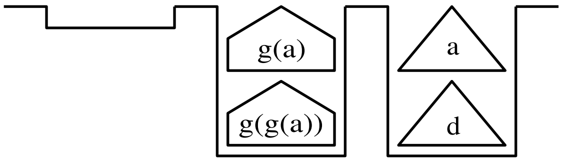
We will describe the rewriting process starting just after the symbol `f` is popped of the control stack. At that moment the traversal stack is empty and the argument stack contains references to the arguments of function `f`.



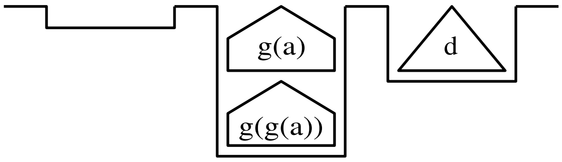
Step 1. The outermost function symbol of the term on top of AS matches with the indicated `g` in $\texttt{f(}\boxed{\texttt{g}}\texttt{(g(b)),c)}$. Consequently, the term is moved to the traversal stack and its arguments are pushed on the argument stack.



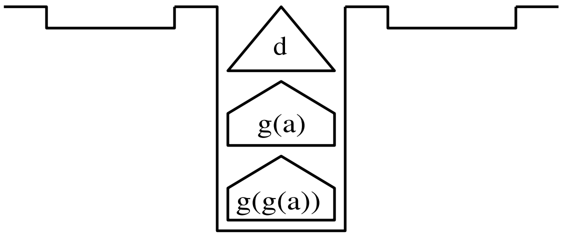
Step 2. The outermost function symbol of the term on top of AS matches with the indicated `g` in $\texttt{f(g(}\boxed{\texttt{g}}\texttt{(b)),c)}$. Consequently, the term is moved to the traversal stack and its arguments are pushed on the argument stack.



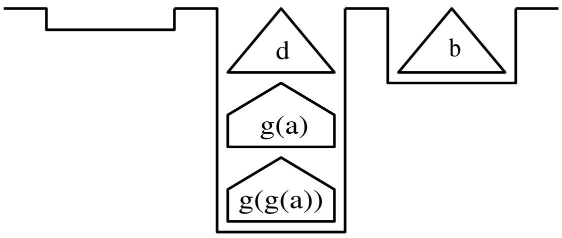
Step 3. The outermost function symbol of the term on top of AS (`a`) does not match with the `b` in rule 1. Consequently, the arguments of the previously recognized function are removed from the argument stack (backtracking).



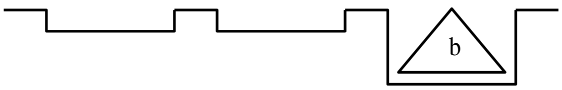
Step 4. The matching process continues from the position immediately after recognition of symbol g and skipping its arguments to the traversal stack. The reference to `g(a)` remains on the traversal stack to represent the value for X. Symbol `d` on top of AS is accepted so it is moved to the traversal stack.



Step 5. The complete left-hand side of rule 2 is recognized. Consequently, the right hand side of that rule is traversed in pre-order and it is pushed on on the argument stack. In this case, this results in the creation of node `b`.



Step 6. The references on the traversal stack produced during matching of `f` are removed (bookkeeping).



Step 7. The argument stack holds a reference to the normalized term for `f(g(g(a)),d)` and rewriting terminates since the control stack is empty.

B.3 The μARM language

In the μARM language, functions are assumed to have at most two arguments. A simple construct is provided to combine arguments in pairs which allows functions to have more than two arguments. This is sufficient for most purposes and it allows for easy memory management. In general, it does not lead to inefficiencies since the overhead introduced by the extra construct is out-weighted by the speed-up in memory management. This restriction is only of importance in the internal working of μARM; the front end of the compiler translates arbitrary TRSs into TRSs in which the functions have their arguments combined in pairs.

Below is a list of all μARM constructs; 'lexical notions' are Sym (any identifier), Num ([0-9]+) and Idx (zero, one or two). In the context of μARM, a trie (pronounced as try) is a (part of a) μARM program. It contains one or more μARMstatements and is typically used to denote a composition of instructions that can be treated as one instruction (for example the instructions that have to be executed when a `match` succeeds).

In this overview we will talk of the stacks as if they contain (sub-)terms. This is not completely the case since the stacks actually hold references to trees on the heap that represent these terms. We make this simplification to keep the overview as clear as possible.

B.3.1 Declarations

start(Sym, Trie1, Trie2)

Trie1 is the definition of the μARM program reducing Sym. Trie2 may contain further definitions (i.e., the rest of the program).

notrie

indicates the end of a list of nested 'start' constructs.

B.3.2 Unconditional flow of control

goto(Sym)

start a new μARM cycle, using Sym.

recycle

start a new μARM cycle, taking the next symbol from the control stack.

B.3.3 Matching

dock(Trie)

After a dock, a sequence of matches follow, which will look at the outermost function symbol (ofs) of the term on the top of the argument stack. Dock stores this ofs in an auxiliary register X.

match(Sym, Trie1, Trie2)

if Sym is equal to the ofs of the term on top of AS (which is in register X) then move the term from the top of the argument stack to the traversal stack, push its arguments on the argument stack and continue with Trie1. Otherwise continue with Trie2.

pairskip(Trie)

the top of the argument stack is a PAIR node. Skip this node to the traversal stack and push its arguments on the argument stack. Continue with Trie.

B.3.4 Stack manipulation

cpush(Sym, Trie)

push Sym on the control stack and continue with Trie.

c2a(Trie)

move the top-node of the control stack to the argument stack and continue with Trie.

cpshv(Num, Trie)

push the n-th entry of the traversal stack on the control stack and continue with Trie.

apshv(Num, Trie)

push the n-th entry of the traversal stack on the argument stack and continue with Trie.

adrop(Num, Trie)

drop Num items from the argument stack and continue with Trie.

tdrop(Num, Trie)

drop Num items from the traversal stack and continue with Trie.

skip(Trie)

move the top node of the argument stack to the traversal stack and continue with Trie.

retract(Trie)

move the top node of the traversal stack to the argument stack and continue with Trie.

B.3.5 Building nodes

ccbuild(Sym, Idx, Trie)
create a new node on the control stack with ofs Sym and Idx arguments. The arguments are taken from the control stack (top is last). Continue with Trie.

acbuild(Sym, Idx, Trie)
create a new node on the argument stack with ofs Sym and Idx arguments. The arguments are taken from the control stack (top is last). Continue with Trie.

atbuild(Sym, Idx, Trie)
create a new node on the argument stack with ofs Sym and Idx arguments. The arguments are taken from the traversal stack (top is last). Continue with Trie.

aabuild(Sym, Idx, Trie)
create a new node on the argument stack with ofs Sym and Idx arguments. The arguments are taken from the argument stack (top is first). Continue with Trie.

B.4 A hybrid interpreter for μARM

A interpreter for μARM has been constructed. The interpreter reads a μARM program and one or more subject terms, and normalizes the terms using that program.

Clearly, interpretation has a drawback: the execution efficiency is reduced due to an interpretational overhead. To some degree this is the price to pay for a short turn-around time, but in certain cases it is unacceptable. Consider for example a library of TRS modules: these modules will rarely change, and the disadvantage of expensive compilation seems unimportant.

For this reason the μARM interpreter is constructed in such a manner that functions which are expected to remain stable can be compiled to machine executable functions, and can be linked to the interpreter to be executed instead of an interpreted function definition. This technique, the combination of compiled and interpreted code, is sometimes referred to as 'melting ice technology'. We will refer to this capability as *hybrid*.

The same mechanism used for linking in compiled functions can also be used to access functions which perform actions that are inexpressible in a term rewriting system, such as I/O functions.

Incrementality imposes some requirements on the hybrid interpreter: if a compiled function is at some point in time changed in the TRS, then it must be possible to replace the compiled definition by an interpreted definition. Alternatively, if the user declares some module to be stable, then its function definitions could be compiled to achieve greater efficiency, and the interpreter should replace the interpreted definitions by the compiled definitions. Consequently, the implementation of a function (as an interpreted or an external function) should only be visible in the definition of that function, and should have no further implications (e.g., there shouldn't be any differences in calling conventions or return values).

A drawback is that dynamic linking is generally not supported on UNIX systems, and there are no ready-to-use portable tools available to achieve it. Consequently, for the time being, interpreted functions cannot become externals and externals cannot receive a new compiled definition at run-time. Until a portable dynamic linking package is available, this behaviour can be reached by rebuilding the interpreter.

Appendix C

An incremental front end for conditional TRSs

C.1 Introduction

In this appendix we will describe the incremental front end of the ASF+SDF compiler. The front end translates conditional term rewriting systems into code for the abstract rewriting machine (see Appendix B on page 47). The front end is incremental: individual functions can be recompiled independently, and only in a few cases does the change of a rewrite rule necessitate recompilation of more than one function.

The compilation process has three stages: the first stage is the translation of a single rule into a *thread*, in which all individual machine actions required to verify and apply the rule are represented.

The second stage involves the combination of all threads pertaining to the same function, and choosing, where necessary, an ordering of rules. That is, the front end implements a specificity ordering: more specific rules have higher priority that less specific rules. For example, the rule $f(g(x), y) = ...$ has precedence over $f(x, y) = ...$, and the non-linear equation $f(x, x) = ...$ has higher precedence than $f(x, y) = ...$ because of the implicit condition.

In addition, the second stage takes care of failure handling: if some attempted match or condition fails, alternatives may still exist and code is generated to inspect these.

The third stage applies global knowledge. In particular the fact whether a function is defined by any rule, or whether it is a free constructor (no rule pertaining to that function exists). If a function is free that fact can be used to generate more efficient code. Note that there is a downside: if a function is initially free, but is subsequently defined by some rule, then all function who's code depends on the constructor being free have to be re-compiled. In practice this situation rarely occurs, so we will accept the occasional overhead.

C.2 Construction of threads (Stage I)

The first stage is the translation of all single rules into threads. A thread contains all 'machine' actions needed to apply a certain rule. Initially some matching actions will be done. Subsequently, conditions are evaluated, and further matching occurs. Finally an instantiation follows.

The entire instantiation process relies on global information, so this part of the thread just carries the term to be instantiated. Similarly, a conditional instruction corresponds to a single term that must be instantiated, and is followed by matching code which inspects the result of the evaluation.

Conditions: μARM does not provide any built-in functionality for conditions. Instead, the concepts of instantiation and matching are used. In addition, auxiliary functions are introduced. Let's consider an example: consider the condition $x * x = x + x$. In μARM this corresponds to the thread $cond(\_0\_, eq(p(0,0), t(0,0)), match(true), ...)$. Here, $\_0\_$ is the auxiliary function symbol generated by the front end for this condition. The term $eq(p(0,0), t(0,0))$ is the term that needs to be instantiated and normalized to evaluate the condition, where 0 is a reference to the location of variable x). If the result matches $true$, the condition holds.

Given a rule, the thread is constructed by traversing the rule in pre-order, visiting the left-hand side first, and then the conditions (in textual order) and finally the right-hand side.

$start(Sym, Thread)$	$\rightarrow$ $Thread$	start of thread
$match(Sym, Thread)$	$\rightarrow$ $Thread$	match current subject against Sym
$skip(Thread)$	$\rightarrow$ $Thread$	skip current subject (i.e. match with variable)
$pairskip(Thread)$	$\rightarrow$ $Thread$	like *match* except symbol is *PAIR*
$inst(Term, Nat)$	$\rightarrow$ $Thread$	instantiate term
$cond(Sym, Term, Thread)$	$\rightarrow$ $Thread$	see conditions described above

Table C.1 Threads in the front end of the compiler.

C.3 Tries and tree automata

C.3.1 Tries

The second and third stages of the front end produce *Tries*. The relation with commonly known tries (a special kind of tree) is lost, but the word 'trie' is still applicable as an acronym for 'Term Recognizing and Instantiating Entity'.

As mentioned, μARM-code is exclusively concerned with matching and instantiation. The matching (recognizing) parts are proper tree matching automata (discussed below). The instantiating parts are sequences of machine instructions, which are dissimilar to automata. As a compromise, the combination is called 'entity'.

In the sequel, we will use common representations and terminology used for automata for tries as well. For example: a trie is a collection of nodes (or states) connected by edges (or transitions); nodes are often labelled with the μARM instruction they correspond with.

Note that tries are more limited than arbitrary automata. A trie is tree- or dag-structured and does not contain a cycle.

C.3.2 Tree automata

All threads relating to a function are combined into a *tree automaton* (*cf.* [Wal91]). A tree automaton is capable of recognizing instances of a potentially large set of tree patterns. Common prefixes of tree patterns are matched against a subject tree only once. For example, the patterns $f(a)$ and $f(b)$ are recognized by the automaton in Figure C.1.

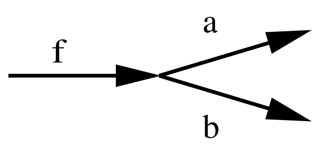


Figure C.1 Automaton for the recognition of $f(a)$ and $f(b)$.

Note that given a term $f(.)$ the automaton will inspect the symbol f only once, whether the term is an instance of the first or the second pattern. It leads too far for this report to discuss tree pattern matching in any further detail.

C.4 Construction of tries (Stage II)

The front end combines all threads relating to the same function symbol into a trie. For most kinds of threads, a related trie-kind exists in which those threads are combined. Internally the front end defines an operator '`+>`' which adds a thread to a trie. As a trivial example consider the following equation taken from the front end:

```
match(Sym,Thread) +> match(Sym,Trie1,Trie2) =
                           match(Sym,Thread+>Trie1,Trie2)
```

This equation states that adding a match-thread to a match-trie for the same symbol, means adding the remainder of the thread to the trie handling the success-case of the match-trie.

The only nontrivial aspect of this stage is the following: after a shared prefix the thread and the trie into which it is added differ, and a choice must be made which action is performed first (it is irrelevant which action occurs in the thread and which in the trie, since the textual ordering of rules is insignificant). The kind of action determines its priority according to the following rule: *condition* > *match* > *skip* > *inst*. That is, conditions are always evaluated first (hence, the implicit equality in non-linear equations is verified before further matching); matching is attempted before *indifferent* matching (i.e., matching against a variable, where the actual sub-term is not inspected); and this is again done before instantiating. The following remarks are in order:

- these priorities apply only to actions in different threads at the end of a shared prefix. They can not influence the order of actions within one thread.
- the rules above roughly define the relative priority of rewrite rules by defining the specificity (partial) order: conditions are more specific than no conditions and explicit function symbols (leading to matching is more specific than variables, where the visiting order is a pre-order traversal of the left-hand side and the conditions. Note that the latter means that $f(g(x), y)$ is more specific than $f(x, g(y))$.
- not all combination defined by the priority rule above actually occur. For example, there is no situation where two thread have a match and an instantiation immediately after a common prefix.
- the priority rule shown doesn't define a total order. For example, two thread with a common prefix followed by two (different) conditions are unordered. The front end chooses an order at random, such that in the resulting trie all actions are totally ordered.

C.4.1 Failures

After combination of all threads, failure-code is added. That is, for every match instruction the failure part is directed towards the next alternative in the (now total) order of combined threads.

The construction of failure code is the most intricate part of the front end. The reason is that some trie may effectively occur as failure-handler for several other case. Consider for example the patterns $f(a,b,c)$, $f(a,x,b)$ and $f(x,y,z)$. The trie handling the most general case ($f(x,y,z)$) is used as failure if no a is found at the first argument, but also if no b is found in the third argument, once the first pattern is rejected. This is easily seen in Figure C.2.

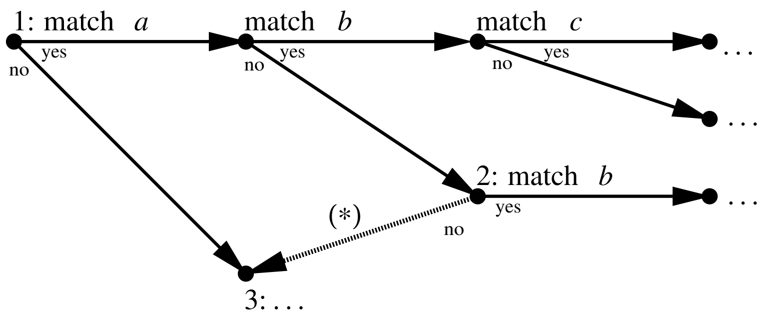


Figure C.2 Automaton for the recognition of $f(a,b,c)$, $f(a,x,b)$ and $f(x,y,z)$.

At run-time, the μARM machine states in node 1 and 2 obviously differ. In node 3, however, the machine state must be the same, independent of how that node was reached, in order to set up correctly for the subsequent code. Therefore, the edge marked $(*)$ is special: bookkeeping actions are performed to change the machine state of node 2 into the machine state of node 1.

The front end maintains a symbolic description of the entire machine state. For every possible pair of states (α,β), it is able to produce the code required to change from state α to state β.

One of the two major reasons to introduce the traversal stack into μARM (as described in Appendix B) is this failure code: in order to get from some state α to its failure state β, terms that have already been visited may again be required. Since the failure state (β) is at most one edge removed from a state on the path to α, the μARM machine state can be reconfigured by a sequence of stack drops (from AS or TS) and stack moves (from TS to AS). An example of the use of the traversal stack is presented in Section B.2.2 on page 50.

C.5 Generation of term instantiating code (Stage III)

The final stage of the front end generates all the term instantiation code. Naively, the to-be-instantiated term could be traversed in pre-order, and all function symbols could be put on the control stack. Several optimizations on this naive approach can be considered:

- Variables correspond to values which are already in normal form. Accordingly, their values aren't traversed, but can be used as-is. This optimization was already applied in ARM [KW93] and is very common.

- Function symbols which are free (i.e., no rules are defined for them), should preferably not be analyzed in subsequent μARM cycles. For example, if g is a free constructor, then pushing g and the value of x (for the pattern $g(x)$) on the control stack leads to a (slight) inefficiency: after the current cycle, the statically known fact that g is free is lost, and must be re-established at run-time.

  This situation can be improved by generating code which, instead of pushing g and the value of x on the control stack, builds a term with g as function symbol and the value of x as only child.

- When items are pushed in pre-order on the control stack, the last items are typically one or more variable values. At the beginning of the next cycle these values will then be moved immediately to the argument stack. This can be improved by generating code which pushes those values immediately on the argument stack.

- If the last item pushed on the control stack is a defined function symbol, the a minor improvement can be achieved by jumping immediately to the definition of that function, instead of pushing the symbol and retrieving it at the beginning of the next cycle. This is a very mild case of *matchless rewriting*: the intermediate function symbol is never matched against.

Observe that the three optimizations above are mutually exclusive and complete. Hence, every pattern instantiation will improve from these optimization.

C.5.1 Instantiating code

Based on the three identified optimizations, the following general framework can be used for generation of term instantiating code: Consider a pattern P. From the leaves upward, we will color all nodes using the following rules:

1. If the node is a variable, color it *green*.

2. if the node is a nullary free constructor, color it *green*.

3. if the node is has free constructor with one or more children, and if all the children are colored green, then color the node itself green.

4. otherwise, color the node red.

After coloring all nodes as indicated the following rules must now be applied:

5 color the right-most inner-most red node *orange*.

6 color all nodes that appear in a pre-order traversal after the orange node *blue*.

The result is a pattern as presented in Figure C.3:

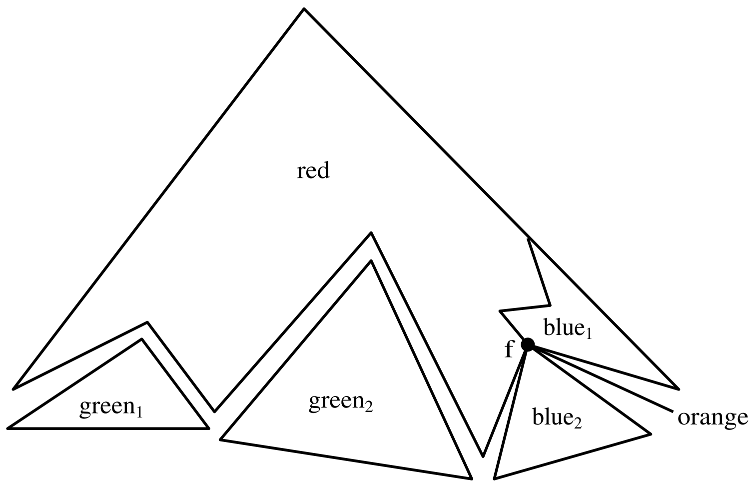


Figure C.3 Colored pattern

Guided by a pre-order traversal until the orange node is encountered, the following code is generated for colored patterns:

- in a red node, generate an instruction that pushes the corresponding function symbol on the control stack.

- in a green sub-pattern, generate code that builds a tree bottom-up, right-to-left. Values of variables are pushed on the control stack; constructors are built taking their arguments from the control stack. For example, the sub-pattern $f(g(x), h(y))$ would produce something like:
  $cpsh(y, ccbuild(h, 1, cpsh(x, ccbuild(g, 1, ccbuild(f, 2)))))$.

Now, all blue patterns sprouting from the path to the orange node are visited from right to left, and for each code is generated that builds the pattern on the argument stack, as for green patterns, bottom-up and right-to-left.

Finally the orange node is processed by generating a *goto* instruction to the function symbol for that node. If no orange node exists, the sequence of instructions for green patterns is concluded with the *recycle* instruction.

push red symbols till root of $green_1$ on CS (in pre-order)
build $green_1$ on CS
push *red* symbols between $green_1$ and $green_2$ on CS
build $green_2$ on CS
push remaining *red* on CS
build $blue_1$ on AS
build $blue_2$ on AS
`goto` f

Table C.2 Outline of the instantiating code for the colored pattern.

C.6 Incrementality

In this Section we will discuss various considerations that have and influence the incremental behaviour of the compiler and of the degree of incrementality (the *granularity*).

C.6.1 Stage I

In Stage I a rule is translated into a thread. There are two reasons to choose the granularity as large as a rule:

- the front-end editor uses this grain size, partly because smaller granularity changes often represent instable (incorrect) intermediate results of edit operations.
- the amount of work needed to process a single rule is small. This must be set off against the additional bookkeeping needed if a smaller size is taken. It is unclear if efficiency would be gained, but it is clear that the difference would not be substantial.

C.6.2 Stage II

In Stage II all threads pertaining to one function are combined into a trie. In addition, a function may become defined after having been free (a rule for it is added), or may become free after having been defined (the last rule for it was deleted).

The new trie is compiled further (by Stage III) and, if the free/defined status of some symbol changed, all tries which depended on this information are also re-compiled.

Note that if a symbol becomes free after having been defined, it is not strictly necessary to recompile functions. In this situation recompilation only improves the efficiency.

Modularity: A word on the role of function definitions versus modularity is in order here. A module is a collection of rules which are conceptually closely related in the underlying specification. In practice, a function is often defined entirely within one module. However, in the algebraic paradigm this is not necessary (nor is it desirable). That is, a function may also be defined distributed over many modules.

A consequence of this is that there is no notion of pre-compiled modules (e.g., a library). It is, however, possible to pre-compile functions that will remain stable, and due to the observation above that functions are usually defined within one module this leads to the same result.

C.6.3 Stage III

The incremental behavior in Stage III is straightforward: after any change recompilation of one or a few function definitions takes place. That is, the grain size for Stage III is a function definition rather than a rule. There is no reason to use a larger grain size than this, although in a production version of the front end it may be desirable to combine several functions together (for example, all function defined in one module). This has no essential effect on the front end.

Appendix D

Code Generator Generators

D.1 Introduction

An automatic generator of compiler parts is known as "compiler-compiler". Techniques for the generation of scanners and parsers have become well known and are widely used while other compiler parts, such as the "back end", which generates the actual machine code, are still mainly written by hand. A back end generator or *code generator generator* (CGG) is a system that automatically builds these code generators. The technique of automatic generation of code generators has been used in research for several years. The code generator is described by a relatively short declarative specification called the *code generator description* (CGD). A back end generator reads this description and automatically builds a code generator.

Figure D.1 gives an overview of code generation using an automatically *generated code generator* (GCG). The front end translates the source program into an intermediate representation (IR). The GCG translates this IR into the target code (i.e., machine code).

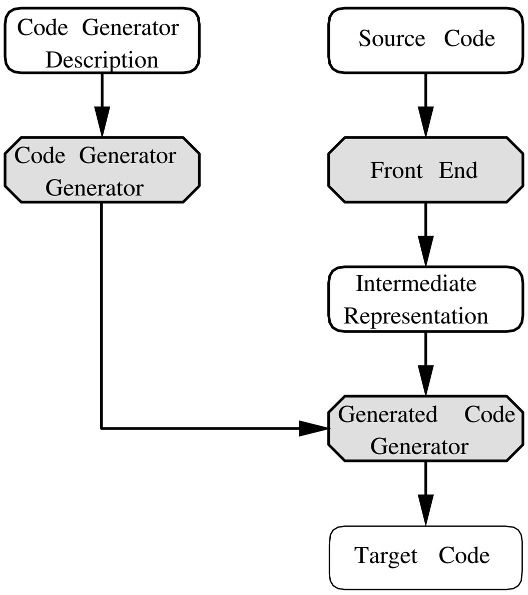


Figure D.1 Code generation using an automatically generated back end.

Developing a code generator by giving a declarative description has several advantages compared to programming the code generator by hand. First it is *easier* and much *less work*. So it is possible to write a description in shorter time with *less effort*. The back end generator can perform consistency checks on the description and hence detect many errors. Therefore an automatically produced back end is *more reliable*. Since the back end is the main machine-dependent part of a well designed compiler, the ability to quickly build reliable back ends allows one to build *portable* compilers.

D.2 The GMD Back End Generator

A now commercially available code generator generator which started as a public domain product is the GMD Back End Generator (`BEG`). The internal algorithms for this system are described in [ESL89]. The use of `BEG` is described in [Emm90].

D.2.1 Code generation using BEG

The `BEG` system automatically generates a code generator out of a code generator description. The input language for this GCG is an intermediate language based on expression trees. Consequently, the front end should translate the source program into a sequence of expression trees. The GCG translates these trees one at a time, so the problem the code generator solves is to produce code for a single expression tree.

One approach to produce code for such an expression tree is to traverse it in postfix order and to emit code for each node separately. This is called *naive code generation*. However, this results in rather bad code because there usually exist machine instructions that are so powerful that they can be used to implement several nodes of the tree at once.

The approach used in `BEG` is to:

1. describe the machine instructions of a given architecture by tree patterns.

2. find a cover of the input tree.

3. emit the machine instructions that correspond to this cover.

This approach allows the use of powerful machine instructions to implement several nodes of the input tree at once, so better (i.e., smaller and more efficient) code is emitted.

A `BEG` code generator description consists of rules, each describing a tree pattern and the corresponding machine instruction. The code generator uses these rules to translate the given input tree to machine code. The first and most important step is to find a *cover* of the input tree. A cover is a collection of rules that describes the complete input tree. Once such a cover is found, the tree has to be traversed in postfix order and the instructions specified in the corresponding rule should be emitted. Note that registers have to be allocated before the instructions are emitted. This is done by the so called *register allocator* which is discussed in Subsection D.2.2. The main problem is the determination of covers. Fortunately this problem is completely solved by `BEG`, the user only has to specify the rules.

There might, and usually will, exist a lot of different covers. Each cover corresponds to a correct translation, but the code quality may vary. So `BEG` should select a cover which corresponds to good code. Therefore a cost value has to be specified for each rule in the code generator description. Usually one takes a weighted sum of execution time and memory requirements of the resulting instruction. The cost of a cover is the sum of the costs of all rules in it. `BEG` selects a cover of minimal cost (a so called *minimal cover*) out of all covers that can be made from patterns in the code generator description. This has some strong implications for the development of that CGD:

- the user only has to describe a set of *possible* translations; `BEG` manages to select the optimal one according to the cost values specified by the user.
- it is not necessary for the user to know any details about the algorithm which determines these minimal covers. It is sufficient to view `BEG` as a *black box* which efficiently determines a minimal cover.
- most complicated behaviour can be expressed by some relatively simple rules; a pattern which is difficult to describe can be divided into several simple patterns that have the same result.
- adding new (correct) rules can not cause any harm; if they are worse than the rules that were already in the CGD, they won't be used since the costs would go up and if they are better, they are used and the resulting code generator is better.
- it is much easier to assure the correctness of a CGD than of a hand written code generator since a CGD is explicitly divided into (small) patterns for each of which it is easy to assure the correctness. Furthermore, it is easier to assure the completeness of a CGD than of a hand written code generator since one only has to assure that all possible input constructs are listed in one way or another as a pattern in the CGD.

D.2.2 Register Allocation

The *register allocator* is the process that is concerned with the determination of which registers are to be used by the emitted machine instructions. `BEG` supports two different register allocators, the general register allocator and the on-the-fly register allocator. The general register allocator is slower than the on-the-fly register allocator but supports a wider range of target machines. Both register allocators are generated as a part of the generated code generator. It is also possible to use a hand written register allocator (this might be necessary if the target is a stack machines).

D.2.3 Remarks

We can make the following remarks on the use of `BEG` for generating (portable) back ends:

- For every target architecture the user has to create a CGD holding the tree patterns of the IR language and their corresponding assembly code.
- The power of `BEG` lies in the ability to find an optimal cover for expression trees. As a result, `BEG` seems to be more suited to generate back ends for a language on relatively high level (e.g., C or Pascal) since their intermediate code will contain more complex expression trees.

- No optimisations are done over a collection of input expressions (for instance reusing parts of previously emitted code). The generated code generator translates just one expression tree at a time.

D.3 BURG

BURG [FHP92b] is a program that generates a fast tree parser using BURS (Bottom-Up Rewrite System) technology [BDB90]. It accepts a cost augmented tree grammar and emits a C program that discovers in linear time an optimal cover of patterns in the language described by the grammar. BURG has been used to construct fast optimal instruction selectors for use in code generation (e.g., the back end of the lcc compiler).

D.3.1 Input

BURG accepts a tree grammar and emits a BURS tree parser, called BURM. BURG's input grammars are structurally similar to YACC's [Joh86], except that BURG accepts no embedded semantic actions. This is done because there was no format which suited all intended applications. Instead, each rule has a unique, *external rule number*. BURM uses these numbers to report the matching rule to a user-supplied routine, which must implement the desired *semantic actions*. In the case of code generation this would mean emitting the machine instructions for this particular rule.

Rules contain a non-negative costs argument. The cost of a cover is the sum of the costs for all rules applied in that cover. The tree parser finds the cover with minimal cost for a given input tree.

D.3.2 Output

BURM traverses the input tree twice. The first pass or *labeller* runs bottom-up and left-to-right, visiting each node exactly once. Each node is labeled with a state, a single number that encodes all full and partial optimal pattern matches that are possible from this node. The second pass or *reducer* traverses the input tree top-down. It accepts a state label and a *goal non-terminal* — initially the root's state label and the start symbol, — which determine the rule to be applied. As the reducer visits (and possibly revisits) each node, the user-supplied code is called to execute the desired semantic actions.

BURM selects an optimal parse for the input tree without cost computations at compile time. Instead, BURG does the cost computations at compile-compile time, as it builds BURM. Consequently, BURM parses very quickly.

D.3.3 Remarks

We can make the following remarks on the use of BURG for generating (portable) back ends:

- For a given intermediate language a tree grammar needs to be build.
- For every target architecture, the user has to write the routine that takes semantic actions, i.e., emits the machine instructions for a given rule.

A "code generator description" for BURG is the combination of this tree grammar and the routine which does the emitting of the machine instructions for a given rule.

D.4 `iburg`

`iburg` [FHP92a] is a program that reads a `BURG` specification and builds a parser that does its cost computations at compile time. The main advantage is that `iburg` allows dynamic costs while costs should be constants for `BURG`. For example, it can be useful to define the costs as a function of `goto(...)`; this gives high costs for a long branch and low costs for a short branch. In `BURG` these costs have to be estimated.

For a discussion of code generation in combination with `iburg` we refer to the discussion of `BURG` (Section D.3) since they are almost identical.

D.5 Conclusions

There are no ready-to-use code generator descriptions for a given architecture available since these descriptions depend on the chosen intermediate language. This intermediate language is, in most cases a derivant of the input language.

The use of one of the presented code generator generators in our case would mean that we must gain a lot of machine-specific knowledge of our target architectures and then build our own code generator descriptions. Moreover, we do not think that they are able to do significant optimizations since our input language (μARM) is a relatively low level language that has an almost one to one mapping to assembly code. Consequently, we think that the use of one of these code generator generators has little or no benefits to us.

Appendix E

Re-using existing back ends

In this chapter, the re-usability of the back ends for several existing compilers is discussed. The main idea behind re-using back-ends is that most well designed compilers consist primarily of two parts:

1. a language dependent and target independent front end which translates the source code into some kind of *intermediate representation (IR)*.
2. a fairly language independent and target dependent back end which translates the intermediate representation into actual machine code.

When μARM code is translated into the intermediate representation for a given compiler we can use its back ends for code generation. This is a powerful scheme if there is a portable compiler with an intermediate representation that fits acceptably to μARM. All present (and future) ports of that compiler can be used for code generation of our language while we only have to make one (relatively simple) translation.

E.1 The GNU C compiler

After rough inspection of the *Gnu C Compiler* (GCC) [Sta89], one might think of two approaches to re-use its back ends. The first one is to use the *Register Transfer Language* as an interface to the back end, the other is to use the so called *tree interface*.

E.1.1 The Register Transfer Language interface

Most of the work in the back end of GCC is done on an intermediate representation called *Register Transfer Language* (RTL) [MRS90, JM91]. This is code for a (virtual) machine with an infinite number of registers which is used for machine-independent optimization. In this language, the machine instructions to be generated are described in an algebraic form that describes what the instruction does.

The RTL language is inspired by Lisp. It has both an internal form, made up of structures that point to other structures, and a textual form that is used in the machine description and in printed debugging dumps. The textual form uses nested parentheses to indicate the pointers in the internal form.

An approach one might think of, is interfacing to the GCC back end using the textual RTL representation. In our case this would mean a translation from μARM to this form of RTL. However this approach is discouraged in the online GNU C Info pages (Section "Reading RTL"): *"People frequently have the idea of using RTL stored as text in a file as an interface between a language front end and the bulk of GNU CC. This idea is not feasible since GNU CC was designed to use RTL internally only. Correct RTL for a given program is very dependent on the particular target machine and the RTL code does not contain all the information about the program."*

E.1.2 The tree interface

The online GNU C Info pages (Section "Reading RTL") presents an interface to GNU CC: *"The proper way to interface GNU CC to a new language front end is with the "tree" data structure. There is no manual for this data structure, but it is described in the files 'tree.h' and 'tree.def'."*

Since the description in these files is very concise and no other documentation can be found, we will not use this approach.

E.2 The `lcc` compiler

The `lcc` compiler [FH91b, FH91c] is a portable ANSI C compiler which is designed and used at Princeton University and AT&T Bell Laboratories.

The interface between the target independent front end and a target dependent back end is described in [FH91a]. It consists of a few shared data structures, some (very simple) functions and a language for defining *directed acyclic graphs (dag)*. This language is used for the intermediate representation of the source program. Directed acyclic graphs are used to combine common subexpressions in an expression. Like an *abstract syntax tree*, a dag has a node for every subexpression; an interior node represents an operator and its children represent its parameters. The difference between a dag and an abstract syntax tree is that a node in a dag which represents a common subexpression has more than one parent. In the corresponding abstract syntax tree, the common subexpression would be presented as a duplicated subtree.

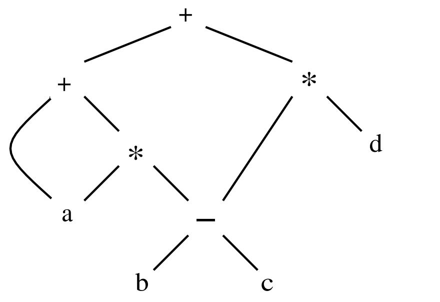


Figure E.1 The directed acyclic graph for $a + a * (b - c) + (b - c) * d$.

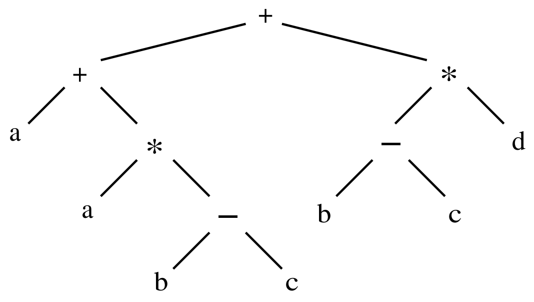


Figure E.2 The abstract syntax tree for $a + a * (b - c) + (b - c) * d$.

A drawback of the `lcc` interface is that the front end and back end are tightly coupled. This approach allows one to build efficient and small compilers but can complicate maintenance since changes in the front end may affect the back end. However, since the interface description is very good and there are several back ends available, the `lcc` compiler might be is a good answer to our code generation problem.

Unfortunately, `lcc` is not public domain software and acquiring a licence is impossible within the time given for our project.

E.3 The VPCC compiler

VPCC (*Very Portable C Compiler*) is a C compiler designed at the University of Virginia. There are back ends available for Sparc, MIPS, 88000, 68000, and 80386 architectures and it is, as its name implies, very easy to port VPCC to other architectures.

The VPCC front end (*cfe*) generates intermediate code for an virtual machine called *CVM*. This code is translated to RTL, optimized and converted to machine code by a back end.

An approach that re-uses the VPCC back ends would involve translation of μARM code into CVM code and process this code with the back end for the desired architecture. A drawback of this approach is that the virtual machine CVM is designed especially as an intermediate stage in the compilation of C code. After close examination of the instruction set of the CVM machine, we feel that μARM programs cannot be expressed efficiently in this language since it is on a much higher abstraction level than μARM. Consequently we will not use the VPCC back ends.

E.4 The SML\NJ compiler

The SML\NJ compiler [AM87] is a Standard ML compiler designed at Princeton University and AT&T Bell Laboratories. Standard ML is a functional language described in [HMM86]. The design of the compiler is highly modular; the front end can be used for other purposes than compilation and the back end is easily retargetable. The compiler consists of three phases:

1. The front end does type-checking and parses a Standard ML program. The output of the front end is a fully typed abstract syntax tree. The front end makes no decisions about runtime locations of variables. It simply assigns them unique identifiers and the back end makes representation decisions.

2. The middle end translates the abstract syntax tree into an expression tree and optimizes the pattern matching.

3. The back end transforms the expression tree into target machine code using a largely machine-independent code generator based on an *abstract machine* interface.

The back end of the SML\NJ compiler can be used for code generation when μARM can be translated into expression trees. However we expect some difficulties doing this since the SML\NJ back end uses closures for data transfer (which is characteristic for functional languages). Another drawback is that the current back ends do not use the full potential of the machine's registers and the user certainly cannot force the use of a register for a given variable. We feel that this may lead to great inefficiencies when generating machine code for μARM since for in that case it is known what variables should be in registers to achieve fast code.

E.5 The Amsterdam Compiler Kit

The Amsterdam Compiler Kit [TvSKS83b] is a collection of programs designed to simplify the task of producing portable (cross) compilers and interpreters. For each language to be compiled a front end must be written to translate the source code into a common intermediate representation. This intermediate code can be optimized and either directly interpreted of translated to the assembly language of the desired target machine by the back end. We considered translating our μARM code into EM, the intermediate code of ACK [TvSKS83a], and then use their back ends. A Drawback to this approach is that there are no back ends available for MIPS architectures and the back end for the SPARC produces rather slow code. Consequently we have decided not to use the Amsterdam Compiler Kit.

E.6 The CORTL intermediate language

CORTL [McC92, McC93] is an intermediate language which is designed to do tree-based code optimization. It is similar to the Register Transfer Language (RTL) used in the GNU C compiler except that CORTL has higher-level control structures.

We will not use CORTL since there are no back ends for this language available.

E.7 Conclusions

We are led to the conclusion that back ends of existing compilers either

- are not usable since they are specifically designed to process an intermediate language derived closely from the source language, or
- are not available for our target architectures

Consequently we will have to find another approach to portable code generation for μARM.

Appendix F

The Virtual Assembly language

In this appendix we will describe the instructions of the Virtual Assembly language (VA). There are five classes:

- Instructions to declare functions and function symbol Id's.
- Instructions for unconditional flow of control.
- Instructions for matching.
- Instructions for stack manipulation.
- Instructions to build nodes on the heap that represent (sub-)terms.

These classes are described in more detail in the sections below.

F.1 Declarations

VA contains a special instruction to declare which function and variable correspond to a given function symbol.

DEFINE(FunSym, Function, FunVar, Id)
: used to declare which function name and variable correspond to a given function symbol (and initialize the variable with the value of Id).

A VA function definition block consists of a header, a number of instructions and a terminator.

FUNCTION(FunName)
: denotes the beginning of a VA function definition block.

END
: denotes the end of a VA function definition block.

F.2 Unconditional flow of control

There are two instructions for unconditional flow of control:

RECYCLE
: starts a new μARM cycle.

GOTO(FunSym)
: continues with the definition of FunSym. This is used to optimize a **RECYCLE** when the next function symbol is statically known.

F.3 Matching

Matching denotes conditional flow of control in a μARM or VA program. If the outermost function symbol of the term on top of the argument stack is the same as a (statically) known function symbol some actions are performed. If not, the next case is examined. When all cases fail, the default action is executed.

There are primarily three instructions that concern matching. These instructions are structured in a similar way as the C `switch` statement. Note however that the syntax is slightly different since unique labels are added to the `case` and `default` instructions to denote the explicit flow of control needed for most assemblers.

SWITCH
: start matching the input term with the outermost function symbol (OFS) of the item on top of the argument stack.

CASE(ThisLabel, Sym, NextLabel)
: denotes the beginning of the instruction block that should be executed when the OFS is equal to Sym. The label "ThisLabel" is the label for this, so called, *CASE instruction block*, "NextLabel" is the label for the following CASE or DEFAULT instruction block.

DEFAULT(ThisLabel)
: denotes the beginning of the instruction block that should be executed when no match for the OFS is found. The label "ThisLabel" is the label for this, so called, *DEFAULT instruction block*.

```
FUNCTION(foo)
  :
SWITCH
CASE(1, a, 2)
  :
RECYCLE or GOTO(...)
CASE(2, b, 3)
  :
RECYCLE or GOTO(...)
DEFAULT(3)
  :
RECYCLE or GOTO(...)
  :
END
```

Figure F.1 Example of VA matching code.

As a result of the design of the μARM language, a CASE or DEFAULT instruction block always ends with a RECYCLE or GOTO(FunSym) instruction. Consequently, there is no need for an instruction that denotes the end of a CASE or DEFAULT instruction block. Furthermore there is always a DEFAULT instruction block, so there is no need for an instruction that denotes the end of a SWITCH block. Figure F.1 gives an example of a VA function that contains code for the matching of a function symbol.

There is an additional instruction for the matching and processing of a PAIR node. This is a special node that is used to allow functions with more that two arguments and it is statically known when this node is on top of the argument stack.

PAIRSKIP
: Move the item on top of the argument stack (which is a PAIR node) to the traversal stack and push its arguments on the argument stack.

F.4 Stack manipulation

PUSH(S)
: creates a new item op top of stack S and returns its location.

TOP(S)
: returns the item on top of stack S.

POP(S)
: returns the item on top of stack S and removes it from S.

MV(V, D)
: Put value V on destination D.

NTH(S, N)
: returns the Nth item on stack S.

DROP(S, N)
: remove N items from stack S.

F.5 Building nodes

BUILD0(FunSym, Dest)
: Build a node having zero arguments and function symbol FunSym on the heap and put a reference to that node on stack Dest.

BUILD1(FunSym, Src, Dest)
: Build a node having one argument and function symbol FunSym on the heap and put a reference to that node on stack Dest. The argument is popped from stack Src.

BUILD12(FunSym, Src, Dest)
: Build a node having two arguments and function symbol FunSym on the heap and put a reference to that node on stack Dest. The arguments are popped from stack Src (the first argument is on top).

BUILD21(FunSym, Src, Dest)
: Build a node having two arguments and function symbol FunSym on the heap and put a reference to that node on stack Dest. The arguments are popped from stack Src (the second argument is on top).

Appendix G

μARM to VA translation

In this chapter we will give the ASF+SDF specification for the translation from a μARM program to Virtual Assembly functions. The translation involves two passes which can be called separately or both at once:

1. In the first pass, a unique ID is generated for every function symbol that is defined in the μARM program. Furthermore, the function and variable that correspond to this symbol are declared.

2. In the second pass, the μARM program is traversed and for every function symbol that is defined, a VA function for the rewriting of this symbol is generated. The rewriting code is deduced from the μARM code in the definition.

The specification consists of 10 modules. The top is the module `BackEnd`, it contains the function to call both passes at once. Directly under the top are two modules (`Symboldefs` and `Functions`). The module `Symboldefs` contains the functions that are used in the first pass, the module `Functions` contains the functions that are used in the second pass.

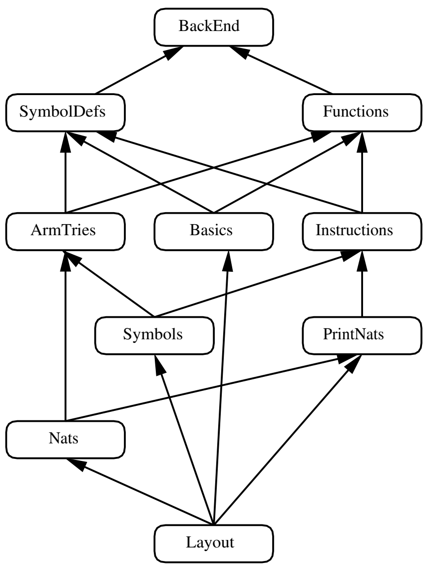


Figure G.1 Import graph of the μARM to VA specification.

The module `ArmTries` contains the definition of the μARM language. The virtual assembly language is defined in the module `Instructions`. Furthermore, there are modules which define function symbols (module `Symbols`), natural numbers (module `Nats`) and functions to print these naturals (module `PrintNats`). The modules `Basics` and `Layout` define some commonly used lexicals and sorts.

Import graphs (i.e., graphs that indicate which modules are imported in which modules) are frequently used during the development of modular algebraic specifications to control their structure and to keep an overview. In Figure G.1 we present the import graph of the specification. The graph presented here is the minimal import graph. This is an import graph that does not show transitively derivable relations (A imports C is transitively derivable when A imports B and B imports C because C is automatically imported in A). The result is a cleaner import graph that is easier to understand. Note that it is allowed in ASF+SDF to import the same module more than once.

G.1 BackEnd

imports Functions(G.3) SymbolDefs(G.2) ArmTries(G.4) Layout(G.10)
exports
  sorts PROGRAM
  context-free syntax
    "*arm2functions*" (Trie) → PROGRAM
    SYMDEFS "/ ∗ ∗ ∗ ∗ ∗ /" FUNCTIONLIST → PROGRAM
hiddens
  variables
    Armprog → Trie

equations

arm2functions(*Armprog*) = [a2f0]
symdefs of Armprog / ∗ ∗ ∗ ∗ ∗ / *functionlist of Armprog*

G.2 SymbolDefs

imports ArmTries(G.4) Instructions(G.5) Nats(G.7) PrintNats(G.8) Basics(G.9) Layout(G.10)
exports
  sorts SYMDEFS
  context-free syntax
    symdefs of Trie → SYMDEFS
    SYMBOLDEF∗ → SYMDEFS
hiddens
  sorts SYMDEFS_INTERNAL
  context-free syntax
    makesymdefs(Trie) → SYMDEFS_INTERNAL
    SYMBOLDEF∗ *x* Nat → SYMDEFS_INTERNAL
  variables
    Armprog [*0-9* ′]∗ → Trie

Rest_of_armprog [*0-9* ′]∗ → Trie
Funsym [*0-9* ′]∗ → Sym
Funarm [*0-9* ′]∗ → Trie
Symboldefs [*0-9* ′]∗ → SYMBOLDEF∗
Next [*0-9* ′]∗ → Nat
Number [*0-9* ′]∗ → Nat

equations

$$\frac{makesymdefs(Armprog) = Symboldefs\ x\ Number}{symdefs\ of\ Armprog = Symboldefs} \quad [SD0]$$

$$\begin{aligned} makesymdefs(notrie) \ = \ & \text{DEFINE}(\texttt{eq}, \texttt{f\_eq}, \texttt{v\_eq}, 0) \\ & \text{DEFINE}(\texttt{true}, \texttt{f\_true}, \texttt{v\_true}, 1) \\ & \text{DEFINE}(\texttt{false}, \texttt{f\_false}, \texttt{v\_false}, 2)\ x\ 3 \end{aligned} \quad [MSD0]$$

$$\frac{makesymdefs(Rest\_of\_armprog) = Symboldefs\ x\ Next}{\begin{array}{l} makesymdefs(start(sym(Ch^*), Funarm, Rest\_of\_armprog)) \ = \\ Symboldefs \\ \text{DEFINE}(sym(Ch^*), sym(\texttt{"f"}\ \texttt{"\_"}\ Ch^*), sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*), print(Next)) \\ x\ Next + 1 \end{array}} \quad [MSD1]$$

G.3 Functions

imports ArmTries(G.4) Instructions(G.5) Nats(G.7) PrintNats(G.8) Basics(G.9) Layout(G.10)
exports
sorts FUNCTIONLIST
context-free syntax
functionlist of Trie → FUNCTIONLIST
FUNCTION∗ → FUNCTIONLIST
makefunction(Sym, Trie) → FUNCTION
hiddens
sorts BODY
context-free syntax
addfunction(FUNCTION, FUNCTIONLIST) → FUNCTIONLIST
makebody(Trie, Nat, Nat) → BODY
INSTRUCTION∗ "*x*" Nat → BODY
is_match(Trie) → YN
variables
Armprog [*0-9* ′]∗ → Trie
Rest_of_armprog [*0-9* ′]∗ → Trie
Function [*0-9* ′]∗ → FUNCTION
Functions [*0-9* ′]∗ → FUNCTION∗

Funsym [*0-9* ′]$*$	$\rightarrow$ Sym
Funarm [*0-9* ′]$*$	$\rightarrow$ Trie
Funbody [*0-9* ′]$*$	$\rightarrow$ INSTRUCTION$*$
Casebody [*0-9* ′]$*$	$\rightarrow$ INSTRUCTION$*$
Functionlist [*0-9* ′]$*$	$\rightarrow$ FUNCTIONLIST
Aantal [*0-9* ′]$*$	$\rightarrow$ Nat
Idx [*0-9* ′]$*$	$\rightarrow$ Nat
CurLab [*0-9* ′]$*$	$\rightarrow$ Nat
NextLab [*0-9* ′]$*$	$\rightarrow$ Nat
ExitLab [*0-9* ′]$*$	$\rightarrow$ Nat

equations

$$\mathit{functionlist\ of\ notrie} = \qquad \text{[fl0]}$$

$$\frac{\mathit{makefunction}(\mathit{Funsym},\ \mathit{Funarm}) = \mathit{Function}}{\begin{array}{l}\mathit{functionlist\ of\ start}(\mathit{Funsym},\ \mathit{Funarm},\ \mathit{Rest\_of\_armprog}) \ = \\ \mathit{addfunction}(\mathit{Function},\ \mathit{functionlist\ of\ Rest\_of\_armprog})\end{array}} \qquad \text{[fl1]}$$

$$\mathit{addfunction}(\mathit{Function},\ ) = \mathit{Function} \qquad \text{[af0]}$$

$$\mathit{addfunction}(\mathit{Function},\ \mathit{Functions}) = \mathit{Function}\ \mathit{Functions} \qquad \text{[af1]}$$

$$\frac{\mathit{makebody}(\mathit{Funarm},\ 1\ 0\ 0,\ 1\ 0\ 0) = \mathit{Funbody}\ x\ \mathit{CurLab}}{\begin{array}{l}\mathit{makefunction}(\mathit{sym}(\mathit{Ch}^*),\ \mathit{Funarm}) \ = \\ \mathrm{FUNCTION}(\mathit{sym}(\texttt{"f"}\ \texttt{"\_"}\ \mathit{Ch}^*))\ \mathit{Funbody}\ \mathrm{END}\end{array}} \qquad \text{[mf0]}$$

$$\begin{array}{l}\mathit{makebody}(\mathit{recycle},\ \mathit{CurLab},\ \mathit{ExitLab}) \ = \\ \mathit{RECYCLE}\ x\ \mathit{CurLab}\end{array} \qquad \text{[mb1]}$$

$$\begin{array}{l}\mathit{makebody}(\mathit{goto}(\mathit{sym}(\mathit{Ch}^*)),\ \mathit{CurLab},\ \mathit{ExitLab}) \ = \\ \mathrm{GOTO}(\mathit{sym}(\texttt{"v"}\ \texttt{"\_"}\ \mathit{Ch}^*))\ x\ \mathit{CurLab}\end{array} \qquad \text{[mb2]}$$

$$\frac{\mathit{makebody}(\mathit{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l}\mathit{makebody}(\mathit{pairskip}(\mathit{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) \ = \\ \mathit{PAIRSKIP}\ \mathit{Funbody}\ x\ \mathit{NextLab}\end{array}} \qquad \text{[mb3]}$$

$$\frac{\mathit{makebody}(\mathit{Rest\_of\_armprog},\ \mathit{CurLab}+1,\ \mathit{CurLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l}\mathit{makebody}(\mathit{dock}(\mathit{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) \ = \\ \mathit{SWITCH}\ \mathit{Funbody}\ x\ \mathit{NextLab}\end{array}} \qquad \text{[mb4]}$$

$$\frac{\begin{array}{c} \mathit{makebody}(\text{Funarm},\ \mathit{CurLab}+1,\ \mathit{ExitLab}) = \mathit{Casebody}\ x\ \mathit{NextLab}_1, \\ \mathit{is\_match}(\text{Rest\_of\_armprog}) = \mathit{yes}, \\ \mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{NextLab}_1,\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}_2 \end{array}}{\begin{array}{l} \mathit{makebody}(\mathit{match}(\mathit{sym}(\mathit{Ch}^*),\ \text{Funarm},\ \text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{CASE}(\mathit{print}(\mathit{CurLab}),\ \mathit{sym}(\texttt{"v"}\ \texttt{"\_"}\ \mathit{Ch}^*),\ \mathit{print}(\mathit{CurLab}+1))\ \mathit{Casebody}\ \mathit{Funbody}\ x\ \mathit{NextLab}_2 \end{array}} \quad \text{[mb5a]}$$

$$\frac{\begin{array}{c} \mathit{makebody}(\text{Funarm},\ \mathit{CurLab}+2,\ \mathit{ExitLab}) = \mathit{Casebody}_1\ x\ \mathit{NextLab}_1, \\ \mathit{is\_match}(\text{Rest\_of\_armprog}) \neq \mathit{yes}, \\ \mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{NextLab}_1,\ \mathit{ExitLab}) = \mathit{Casebody}_2\ x\ \mathit{NextLab}_2 \end{array}}{\begin{array}{l} \mathit{makebody}(\mathit{match}(\mathit{sym}(\mathit{Ch}^*),\ \text{Funarm},\ \text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{CASE}(\mathit{print}(\mathit{CurLab}),\ \mathit{sym}(\texttt{"v"}\ \texttt{"\_"}\ \mathit{Ch}^*),\ \mathit{print}(\mathit{CurLab}+1))\ \mathit{Casebody}_1 \\ \text{DEFAULT}(\mathit{print}(\mathit{CurLab}+1))\ \mathit{Casebody}_2\ x\ \mathit{NextLab}_2 \end{array}} \quad \text{[mb5b]}$$

$$\mathit{is\_match}(\mathit{match}(\text{Funsym},\ \text{Funarm},\ \text{Rest\_of\_armprog})) = \mathit{yes} \quad \text{[im0]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{adrop}(\text{Aantal},\ \text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{DROP}(\mathit{AP},\ \mathit{print}(\text{Aantal}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb6]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{tdrop}(\text{Aantal},\ \text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{DROP}(\mathit{TP},\ \mathit{print}(\text{Aantal}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb7]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{cpshv}(\text{Idx},\ \text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{MV}(\text{NTH}(\mathit{TP},\ \mathit{print}(\text{Idx})),\ \text{PUSH}(\mathit{CP}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb8]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{apshv}(\text{Idx},\ \text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{MV}(\text{NTH}(\mathit{TP},\ \mathit{print}(\text{Idx})),\ \text{PUSH}(\mathit{AP}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb9]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{skip}(\text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{MV}(\text{POP}(\mathit{AP}),\ \text{PUSH}(\mathit{TP}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb10]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{retract}(\text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{MV}(\text{POP}(\mathit{TP}),\ \text{PUSH}(\mathit{AP}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb11]}$$

$$\frac{\mathit{makebody}(\text{Rest\_of\_armprog},\ \mathit{CurLab},\ \mathit{ExitLab}) = \mathit{Funbody}\ x\ \mathit{NextLab}}{\begin{array}{l} \mathit{makebody}(\mathit{c2a}(\text{Rest\_of\_armprog}),\ \mathit{CurLab},\ \mathit{ExitLab}) = \\ \text{MV}(\text{POP}(\mathit{CP}),\ \text{PUSH}(\mathit{AP}))\ \mathit{Funbody}\ x\ \mathit{NextLab} \end{array}} \quad \text{[mb12]}$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(ccbuild(sym(Ch^*),\ 0,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD0(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ CP)\ Funbody\ x\ NextLab \end{array}} \quad [mb13a]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(ccbuild(sym(Ch^*),\ 1,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD1(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ CP,\ CP)\ Funbody\ x\ NextLab \end{array}} \quad [mb13b]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(ccbuild(sym(Ch^*),\ 2,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD21(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ CP,\ CP)\ Funbody\ x\ NextLab \end{array}} \quad [mb13c]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(acbuild(sym(Ch^*),\ 0,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD0(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb14a]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(acbuild(sym(Ch^*),\ 1,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD1(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ CP,\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb14b]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(acbuild(sym(Ch^*),\ 2,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD21(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ CP,\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb14c]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(atbuild(sym(Ch^*),\ 0,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD0(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb15a]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(atbuild(sym(Ch^*),\ 1,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD1(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ TP,\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb15b]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(atbuild(sym(Ch^*),\ 2,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD21(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ TP,\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb15c]$$

$$\frac{makebody(Rest\_of\_armprog,\ CurLab,\ ExitLab) = Funbody\ x\ NextLab}{\begin{array}{l} makebody(aabuild(sym(Ch^*),\ 0,\ Rest\_of\_armprog),\ CurLab,\ ExitLab)\ = \\ BUILD0(sym(\texttt{"v"}\ \texttt{"\_"}\ Ch^*),\ AP)\ Funbody\ x\ NextLab \end{array}} \quad [mb16a]$$

$$\frac{\textit{makebody}(\textit{Rest\_of\_armprog}, \textit{CurLab}, \textit{ExitLab}) = \textit{Funbody}\ x\ \textit{NextLab}}{\textit{makebody}(\textit{aabuild}(\textit{sym}(\textit{Ch}^*), 1, \textit{Rest\_of\_armprog}), \textit{CurLab}, \textit{ExitLab}) = \textit{BUILD1}(\textit{sym}(\texttt{"v" "\_"}\ \textit{Ch}^*), \textit{AP}, \textit{AP})\ \textit{Funbody}\ x\ \textit{NextLab}} \quad \text{[mb16b]}$$

$$\frac{\textit{makebody}(\textit{Rest\_of\_armprog}, \textit{CurLab}, \textit{ExitLab}) = \textit{Funbody}\ x\ \textit{NextLab}}{\textit{makebody}(\textit{aabuild}(\textit{sym}(\textit{Ch}^*), 2, \textit{Rest\_of\_armprog}), \textit{CurLab}, \textit{ExitLab}) = \textit{BUILD12}(\textit{sym}(\texttt{"v" "\_"}\ \textit{Ch}^*), \textit{AP}, \textit{AP})\ \textit{Funbody}\ x\ \textit{NextLab}} \quad \text{[mb16c]}$$

$$\frac{\textit{makebody}(\textit{Rest\_of\_armprog}, \textit{CurLab}, \textit{ExitLab}) = \textit{Funbody}\ x\ \textit{NextLab}}{\textit{makebody}(\textit{cpush}(\textit{sym}(\textit{Ch}^*), \textit{Rest\_of\_armprog}), \textit{CurLab}, \textit{ExitLab}) = \mathrm{MV}(\textit{sym}(\texttt{"v" "\_"}\ \textit{Ch}^*), \mathrm{PUSH}(\textit{CP}))\ \textit{Funbody}\ x\ \textit{NextLab}} \quad \text{[mb17]}$$

G.4 ArmTries

imports Layout(G.10) Symbols(G.6) Nats(G.7)
exports
  sorts Trie
  context-free syntax

skip "(" Trie ")"	→ Trie
retract "(" Trie ")"	→ Trie
pairskip "(" Trie ")"	→ Trie
notrie	→ Trie
adrop "(" Nat "," Trie ")"	→ Trie
start "(" Sym "," Trie "," Trie ")"	→ Trie
match "(" Sym "," Trie "," Trie ")"	→ Trie
tdrop "(" Nat "," Trie ")"	→ Trie
dock "(" Trie ")"	→ Trie
recycle	→ Trie
goto "(" Sym ")"	→ Trie
cpush "(" Sym "," Trie ")"	→ Trie
cpshv "(" Nat "," Trie ")"	→ Trie
apshv "(" Nat "," Trie ")"	→ Trie
c2a "(" Trie ")"	→ Trie
ccbuild "(" Sym "," Nat "," Trie ")"	→ Trie
acbuild "(" Sym "," Nat "," Trie ")"	→ Trie
atbuild "(" Sym "," Nat "," Trie ")"	→ Trie
aabuild "(" Sym "," Nat "," Trie ")"	→ Trie

hiddens
  variables
    Trie [0-9 ′]*→ Trie
    Nat [0-9 ′]*→ Nat

equations

$$retract(skip(Trie)) = Trie \quad [T10]$$
$$retract(adrop(Nat, Trie)) = adrop(Nat - 1, tdrop(1, Trie)) \quad [T11]$$
$$skip(retract(Trie)) = Trie \quad [T20]$$
$$skip(tdrop(Nat, Trie)) = adrop(1, tdrop(Nat - 1, Trie)) \quad [T21]$$
$$adrop(0, Trie) = Trie \quad [T30]$$
$$adrop(Nat_1, adrop(Nat_2, Trie)) = adrop(Nat_1 + Nat_2, Trie) \quad [T31]$$
$$tdrop(0, Trie) = Trie \quad [T40]$$
$$tdrop(Nat_1, tdrop(Nat_2, Trie)) = tdrop(Nat_1 + Nat_2, Trie) \quad [T41]$$

G.5 Instructions

imports Symbols(G.6) PrintNats(G.8)
exports
sorts SYMBOLDEF FUNCTION INSTRUCTION STACKPOINTER ADRESS VALUE
context-free syntax

"FUNCTION" "(" Sym ")" INSTRUCTION∗ "END" → FUNCTION

"DEFINE" "(" Sym "," Sym "," Sym "," PNAT ")" → SYMBOLDEF

"PUSH" "(" STACKPOINTER ")" → ADRESS
"TOP" "(" STACKPOINTER ")" → VALUE
"POP" "(" STACKPOINTER ")" → VALUE
"NTH" "(" STACKPOINTER "," PNAT ")" → VALUE

"MV" "(" VALUE "," ADRESS ")" → INSTRUCTION
"DROP" "(" STACKPOINTER "," PNAT ")" → INSTRUCTION
"PAIRSKIP" → INSTRUCTION

"*BUILD0*" "(" Sym "," STACKPOINTER ")" → INSTRUCTION
"*BUILD1*" "(" Sym "," STACKPOINTER "," STACKPOINTER ")" → INSTRUCTION
"*BUILD12*" "(" Sym "," STACKPOINTER "," STACKPOINTER ")" → INSTRUCTION
"*BUILD21*" "(" Sym "," STACKPOINTER "," STACKPOINTER ")" → INSTRUCTION

"RECYCLE" → INSTRUCTION
"GOTO" "(" Sym ")" → INSTRUCTION

"SWITCH" → INSTRUCTION
"CASE" "(" PNAT "," Sym "," PNAT ")" → INSTRUCTION
"DEFAULT" "(" PNAT ")" → INSTRUCTION

"AP" → STACKPOINTER
"CP" → STACKPOINTER
"TP" → STACKPOINTER

Sym $\rightarrow$ VALUE

hiddens

variables

Stackpointer$\rightarrow$ STACKPOINTER

equations

$$\mathrm{NTH}(Stackpointer, 0) = \mathrm{TOP}(Stackpointer) \quad [\mathrm{N0}]$$

G.6 Symbols

imports Layout[(G.10)]

exports

sorts Sym

lexical syntax

[*A-Za-z0-9_*]$*$ $\rightarrow$ Sym

variables

Sym [*0-9* $'$]$*$$\rightarrow$ Sym

G.7 Nats

imports Layout[(G.10)]

exports

sorts Nat

lexical syntax

[*0-9*] $\rightarrow$ Nat

context-free syntax

Nat Nat $\rightarrow$ Nat {**left**}

"$-$" Nat $\rightarrow$ Nat

Nat "+" Nat $\rightarrow$ Nat {**left**}

Nat "$-$" Nat $\rightarrow$ Nat {**left**}

"(" Nat ")" $\rightarrow$ Nat {**bracket**}

priorities

{ Nat Nat $\rightarrow$ Nat } > { "$-$"Nat $\rightarrow$ Nat } >

{**left**: Nat "+"Nat $\rightarrow$ Nat, Nat "$-$"Nat $\rightarrow$ Nat}

variables

[*W-Z*][*0-9* $'$]$*$$\rightarrow$ Nat

equations

$$0\ X = X \quad [\mathrm{Nat0}]$$

$$X\,(Y\,Z) \quad = (X + Y)\;Z \qquad \text{[Nat1]}$$
$$X\,(-\;Y\,Z) = -\,(Y - X)\;Z \qquad \text{[Nat2]}$$

$$(-\;X)\;Y = -\;X\,(-\;Y) \qquad \text{[NatU0]}$$
$$-\,-\;X \quad = X \qquad \text{[NatN1]}$$
$$-\;0 \quad = 0 \qquad \text{[NatN0]}$$

$$0 + X \quad = X \qquad \text{[NatP1]}$$
$$X + 0 \quad = X \qquad \text{[NatP2]}$$
$$X\;Y + Z \;= X\,(Y + Z) \qquad \text{[NatP3]}$$
$$X + Y\;Z \;= Y\,(X + Z) \qquad \text{[NatP4]}$$
$$-\;X + Y = Y - X \qquad \text{[NatP5]}$$
$$X + -\;Y = X - Y \qquad \text{[NatP6]}$$

$1\,(-\;1) = 9$ [NatX11]	$2\,(-\;1) = 1\;9$ [NatX21]	$3\,(-\;1) = 2\;9$ [NatX31]
$4\,(-\;1) = 3\;9$ [NatX41]	$5\,(-\;1) = 4\;9$ [NatX51]	$6\,(-\;1) = 5\;9$ [NatX61]
$7\,(-\;1) = 6\;9$ [NatX71]	$8\,(-\;1) = 7\;9$ [NatX81]	$9\,(-\;1) = 8\;9$ [NatX91]
$1\,(-\;2) = 8$ [NatX12]	$2\,(-\;2) = 1\;8$ [NatX22]	$3\,(-\;2) = 2\;8$ [NatX32]
$4\,(-\;2) = 3\;8$ [NatX42]	$5\,(-\;2) = 4\;8$ [NatX52]	$6\,(-\;2) = 5\;8$ [NatX62]
$7\,(-\;2) = 6\;8$ [NatX72]	$8\,(-\;2) = 7\;8$ [NatX82]	$9\,(-\;2) = 8\;8$ [NatX92]
$1\,(-\;3) = 7$ [NatX13]	$2\,(-\;3) = 1\;7$ [NatX23]	$3\,(-\;3) = 2\;7$ [NatX33]
$4\,(-\;3) = 3\;7$ [NatX43]	$5\,(-\;3) = 4\;7$ [NatX53]	$6\,(-\;3) = 5\;7$ [NatX63]
$7\,(-\;3) = 6\;7$ [NatX73]	$8\,(-\;3) = 7\;7$ [NatX83]	$9\,(-\;3) = 8\;7$ [NatX93]
$1\,(-\;4) = 6$ [NatX14]	$2\,(-\;4) = 1\;6$ [NatX24]	$3\,(-\;4) = 2\;6$ [NatX34]
$4\,(-\;4) = 3\;6$ [NatX44]	$5\,(-\;4) = 4\;6$ [NatX54]	$6\,(-\;4) = 5\;6$ [NatX64]
$7\,(-\;4) = 6\;6$ [NatX74]	$8\,(-\;4) = 7\;6$ [NatX84]	$9\,(-\;4) = 8\;6$ [NatX94]
$1\,(-\;5) = 5$ [NatX15]	$2\,(-\;5) = 1\;5$ [NatX25]	$3\,(-\;5) = 2\;5$ [NatX35]
$4\,(-\;5) = 3\;5$ [NatX45]	$5\,(-\;5) = 4\;5$ [NatX55]	$6\,(-\;5) = 5\;5$ [NatX65]
$7\,(-\;5) = 6\;5$ [NatX75]	$8\,(-\;5) = 7\;5$ [NatX85]	$9\,(-\;5) = 8\;5$ [NatX95]
$1\,(-\;6) = 4$ [NatX16]	$2\,(-\;6) = 1\;4$ [NatX26]	$3\,(-\;6) = 2\;4$ [NatX36]
$4\,(-\;6) = 3\;4$ [NatX46]	$5\,(-\;6) = 4\;4$ [NatX56]	$6\,(-\;6) = 5\;4$ [NatX66]
$7\,(-\;6) = 6\;4$ [NatX76]	$8\,(-\;6) = 7\;4$ [NatX86]	$9\,(-\;6) = 8\;4$ [NatX96]
$1\,(-\;7) = 3$ [NatX17]	$2\,(-\;7) = 1\;3$ [NatX27]	$3\,(-\;7) = 2\;3$ [NatX37]
$4\,(-\;7) = 3\;3$ [NatX47]	$5\,(-\;7) = 4\;3$ [NatX57]	$6\,(-\;7) = 5\;3$ [NatX67]
$7\,(-\;7) = 6\;3$ [NatX77]	$8\,(-\;7) = 7\;3$ [NatX87]	$9\,(-\;7) = 8\;3$ [NatX97]
$1\,(-\;8) = 2$ [NatX18]	$2\,(-\;8) = 1\;2$ [NatX28]	$3\,(-\;8) = 2\;2$ [NatX38]
$4\,(-\;8) = 3\;2$ [NatX48]	$5\,(-\;8) = 4\;2$ [NatX58]	$6\,(-\;8) = 5\;2$ [NatX68]
$7\,(-\;8) = 6\;2$ [NatX78]	$8\,(-\;8) = 7\;2$ [NatX88]	$9\,(-\;8) = 8\;2$ [NatX98]
$1\,(-\;9) = 1$ [NatX19]	$2\,(-\;9) = 1\;1$ [NatX29]	$3\,(-\;9) = 2\;1$ [NatX39]
$4\,(-\;9) = 3\;1$ [NatX49]	$5\,(-\;9) = 4\;1$ [NatX59]	$6\,(-\;9) = 5\;1$ [NatX69]
$7\,(-\;9) = 6\;1$ [NatX79]	$8\,(-\;9) = 7\;1$ [NatX89]	$9\,(-\;9) = 8\;1$ [NatX99]

$X\;0\,(-\;1) = X\,(-\;1)\;9$ [NatX01]	$X\;0\,(-\;2) = X\,(-\;1)\;8$ [NatX02]	$X\;0\,(-\;3) = X\,(-\;1)\;7$ [NatX03]
$X\;0\,(-\;4) = X\,(-\;1)\;6$ [NatX04]	$X\;0\,(-\;5) = X\,(-\;1)\;5$ [NatX05]	$X\;0\,(-\;6) = X\,(-\;1)\;4$ [NatX06]
$X\;0\,(-\;7) = X\,(-\;1)\;3$ [NatX07]	$X\;0\,(-\;8) = X\,(-\;1)\;2$ [NatX08]	$X\;0\,(-\;9) = X\,(-\;1)\;1$ [NatX09]

$X 1 (- 1) = X 0 9$ [NatX11] $X 2 (- 1) = X 1 9$ [NatX21] $X 3 (- 1) = X 2 9$ [NatX31]
$X 4 (- 1) = X 3 9$ [NatX41] $X 5 (- 1) = X 4 9$ [NatX51] $X 6 (- 1) = X 5 9$ [NatX61]
$X 7 (- 1) = X 6 9$ [NatX71] $X 8 (- 1) = X 7 9$ [NatX81] $X 9 (- 1) = X 8 9$ [NatX91]
$X 1 (- 2) = X 0 8$ [NatX12] $X 2 (- 2) = X 1 8$ [NatX22] $X 3 (- 2) = X 2 8$ [NatX32]
$X 4 (- 2) = X 3 8$ [NatX42] $X 5 (- 2) = X 4 8$ [NatX52] $X 6 (- 2) = X 5 8$ [NatX62]
$X 7 (- 2) = X 6 8$ [NatX72] $X 8 (- 2) = X 7 8$ [NatX82] $X 9 (- 2) = X 8 8$ [NatX92]
$X 1 (- 3) = X 0 7$ [NatX13] $X 2 (- 3) = X 1 7$ [NatX23] $X 3 (- 3) = X 2 7$ [NatX33]
$X 4 (- 3) = X 3 7$ [NatX43] $X 5 (- 3) = X 4 7$ [NatX53] $X 6 (- 3) = X 5 7$ [NatX63]
$X 7 (- 3) = X 6 7$ [NatX73] $X 8 (- 3) = X 7 7$ [NatX83] $X 9 (- 3) = X 8 7$ [NatX93]
$X 1 (- 4) = X 0 6$ [NatX14] $X 2 (- 4) = X 1 6$ [NatX24] $X 3 (- 4) = X 2 6$ [NatX34]
$X 4 (- 4) = X 3 6$ [NatX44] $X 5 (- 4) = X 4 6$ [NatX54] $X 6 (- 4) = X 5 6$ [NatX64]
$X 7 (- 4) = X 6 6$ [NatX74] $X 8 (- 4) = X 7 6$ [NatX84] $X 9 (- 4) = X 8 6$ [NatX94]
$X 1 (- 5) = X 0 5$ [NatX15] $X 2 (- 5) = X 1 5$ [NatX25] $X 3 (- 5) = X 2 5$ [NatX35]
$X 4 (- 5) = X 3 5$ [NatX45] $X 5 (- 5) = X 4 5$ [NatX55] $X 6 (- 5) = X 5 5$ [NatX65]
$X 7 (- 5) = X 6 5$ [NatX75] $X 8 (- 5) = X 7 5$ [NatX85] $X 9 (- 5) = X 8 5$ [NatX95]
$X 1 (- 6) = X 0 4$ [NatX16] $X 2 (- 6) = X 1 4$ [NatX26] $X 3 (- 6) = X 2 4$ [NatX36]
$X 4 (- 6) = X 3 4$ [NatX46] $X 5 (- 6) = X 4 4$ [NatX56] $X 6 (- 6) = X 5 4$ [NatX66]
$X 7 (- 6) = X 6 4$ [NatX76] $X 8 (- 6) = X 7 4$ [NatX86] $X 9 (- 6) = X 8 4$ [NatX96]
$X 1 (- 7) = X 0 3$ [NatX17] $X 2 (- 7) = X 1 3$ [NatX27] $X 3 (- 7) = X 2 3$ [NatX37]
$X 4 (- 7) = X 3 3$ [NatX47] $X 5 (- 7) = X 4 3$ [NatX57] $X 6 (- 7) = X 5 3$ [NatX67]
$X 7 (- 7) = X 6 3$ [NatX77] $X 8 (- 7) = X 7 3$ [NatX87] $X 9 (- 7) = X 8 3$ [NatX97]
$X 1 (- 8) = X 0 2$ [NatX18] $X 2 (- 8) = X 1 2$ [NatX28] $X 3 (- 8) = X 2 2$ [NatX38]
$X 4 (- 8) = X 3 2$ [NatX48] $X 5 (- 8) = X 4 2$ [NatX58] $X 6 (- 8) = X 5 2$ [NatX68]
$X 7 (- 8) = X 6 2$ [NatX78] $X 8 (- 8) = X 7 2$ [NatX88] $X 9 (- 8) = X 8 2$ [NatX98]
$X 1 (- 9) = X 0 1$ [NatX19] $X 2 (- 9) = X 1 1$ [NatX29] $X 3 (- 9) = X 2 1$ [NatX39]
$X 4 (- 9) = X 3 1$ [NatX49] $X 5 (- 9) = X 4 1$ [NatX59] $X 6 (- 9) = X 5 1$ [NatX69]
$X 7 (- 9) = X 6 1$ [NatX79] $X 8 (- 9) = X 7 1$ [NatX89] $X 9 (- 9) = X 8 1$ [NatX99]

$1 + 1 = 2$ [NatP11] $2 + 1 = 3$ [NatP21] $3 + 1 = 4$ [NatP31]
$4 + 1 = 5$ [NatP41] $5 + 1 = 6$ [NatP51] $6 + 1 = 7$ [NatP61]
$7 + 1 = 8$ [NatP71] $8 + 1 = 9$ [NatP81] $9 + 1 = 1 0$ [NatP91]

$1 + 2 = 3$ [NatP12] $2 + 2 = 4$ [NatP22] $3 + 2 = 5$ [NatP32]
$4 + 2 = 6$ [NatP42] $5 + 2 = 7$ [NatP52] $6 + 2 = 8$ [NatP62]
$7 + 2 = 9$ [NatP72] $8 + 2 = 1 0$ [NatP82] $9 + 2 = 1 1$ [NatP92]

$1 + 3 = 4$ [NatP13] $2 + 3 = 5$ [NatP23] $3 + 3 = 6$ [NatP33]
$4 + 3 = 7$ [NatP43] $5 + 3 = 8$ [NatP53] $6 + 3 = 9$ [NatP63]
$7 + 3 = 1 0$ [NatP73] $8 + 3 = 1 1$ [NatP83] $9 + 3 = 1 2$ [NatP93]

$1 + 4 = 5$ [NatP14] $2 + 4 = 6$ [NatP24] $3 + 4 = 7$ [NatP34]
$4 + 4 = 8$ [NatP44] $5 + 4 = 9$ [NatP54] $6 + 4 = 1 0$ [NatP64]
$7 + 4 = 1 1$ [NatP74] $8 + 4 = 1 2$ [NatP84] $9 + 4 = 1 3$ [NatP94]

$1 + 5 = 6$ [NatP15] $2 + 5 = 7$ [NatP25] $3 + 5 = 8$ [NatP35]
$4 + 5 = 9$ [NatP45] $5 + 5 = 1 0$ [NatP55] $6 + 5 = 1 1$ [NatP65]
$7 + 5 = 1 2$ [NatP75] $8 + 5 = 1 3$ [NatP85] $9 + 5 = 1 4$ [NatP95]

$1+6=7$ [NatP16] $2+6=8$ [NatP26] $3+6=9$ [NatP36]
$4+6=1\,0$ [NatP46] $5+6=1\,1$ [NatP56] $6+6=1\,2$ [NatP66]
$7+6=1\,3$ [NatP76] $8+6=1\,4$ [NatP86] $9+6=1\,5$ [NatP96]

$1+7=8$ [NatP17] $2+7=9$ [NatP27] $3+7=1\,0$ [NatP37]
$4+7=1\,1$ [NatP47] $5+7=1\,2$ [NatP57] $6+7=1\,3$ [NatP67]
$7+7=1\,4$ [NatP77] $8+7=1\,5$ [NatP87] $9+7=1\,6$ [NatP97]

$1+8=9$ [NatP18] $2+8=1\,0$ [NatP28] $3+8=1\,1$ [NatP38]
$4+8=1\,2$ [NatP48] $5+8=1\,3$ [NatP58] $6+8=1\,4$ [NatP68]
$7+8=1\,5$ [NatP78] $8+8=1\,6$ [NatP88] $9+8=1\,7$ [NatP98]

$1+9=1\,0$ [NatP19] $2+9=1\,1$ [NatP29] $3+9=1\,2$ [NatP39]
$4+9=1\,3$ [NatP49] $5+9=1\,4$ [NatP59] $6+9=1\,5$ [NatP69]
$7+9=1\,6$ [NatP79] $8+9=1\,7$ [NatP89] $9+9=1\,8$ [NatP99]

$$0-X \quad = -\,X \tag{NatM1}$$
$$X-0 \quad = X \tag{NatM2}$$
$$X - -\,Y = X + Y \tag{NatM3}$$
$$X\,Y - Z = X\,(Y-Z) \tag{NatM4}$$
$$X - Y\,Z = -\,Y\,(Z-X) \tag{NatM5}$$

$1-1=0$ [NatM11] $2-1=1$ [NatM21] $3-1=2$ [NatM31]
$4-1=3$ [NatM41] $5-1=4$ [NatM51] $6-1=5$ [NatM61]
$7-1=6$ [NatM71] $8-1=7$ [NatM81] $9-1=8$ [NatM91]

$1-2=-\,1$ [NatM12] $2-2=0$ [NatM22] $3-2=1$ [NatM32]
$4-2=2$ [NatM42] $5-2=3$ [NatM52] $6-2=4$ [NatM62]
$7-2=5$ [NatM72] $8-2=6$ [NatM82] $9-2=7$ [NatM92]

$1-3=-\,2$ [NatM13] $2-3=-\,1$ [NatM23] $3-3=0$ [NatM33]
$4-3=1$ [NatM43] $5-3=2$ [NatM53] $6-3=3$ [NatM63]
$7-3=4$ [NatM73] $8-3=5$ [NatM83] $9-3=6$ [NatM93]

$1-4=-\,3$ [NatM14] $2-4=-\,2$ [NatM24] $3-4=-\,1$ [NatM34]
$4-4=0$ [NatM44] $5-4=1$ [NatM54] $6-4=2$ [NatM64]
$7-4=3$ [NatM74] $8-4=4$ [NatM84] $9-4=5$ [NatM94]

$1-5=-\,4$ [NatM15] $2-5=-\,3$ [NatM25] $3-5=-\,2$ [NatM35]
$4-5=-\,1$ [NatM45] $5-5=0$ [NatM55] $6-5=1$ [NatM65]
$7-5=2$ [NatM75] $8-5=3$ [NatM85] $9-5=4$ [NatM95]

$1 - 6 = -\ 5$ [NatM16]	$2 - 6 = -\ 4$ [NatM26]	$3 - 6 = -\ 3$ [NatM36]
$4 - 6 = -\ 2$ [NatM46]	$5 - 6 = -\ 1$ [NatM56]	$6 - 6 = 0$ [NatM66]
$7 - 6 = 1$ [NatM76]	$8 - 6 = 2$ [NatM86]	$9 - 6 = 3$ [NatM96]

$1 - 7 = -\ 6$ [NatM17]	$2 - 7 = -\ 5$ [NatM27]	$3 - 7 = -\ 4$ [NatM37]
$4 - 7 = -\ 3$ [NatM47]	$5 - 7 = -\ 2$ [NatM57]	$6 - 7 = -\ 1$ [NatM67]
$7 - 7 = 0$ [NatM77]	$8 - 7 = 1$ [NatM87]	$9 - 7 = 2$ [NatM97]

$1 - 8 = -\ 7$ [NatM18]	$2 - 8 = -\ 6$ [NatM28]	$3 - 8 = -\ 5$ [NatM38]
$4 - 8 = -\ 4$ [NatM48]	$5 - 8 = -\ 3$ [NatM58]	$6 - 8 = -\ 2$ [NatM68]
$7 - 8 = -\ 1$ [NatM78]	$8 - 8 = 0$ [NatM88]	$9 - 8 = 1$ [NatM98]

$1 - 9 = -\ 8$ [NatM19]	$2 - 9 = -\ 7$ [NatM29]	$3 - 9 = -\ 6$ [NatM39]
$4 - 9 = -\ 5$ [NatM49]	$5 - 9 = -\ 4$ [NatM59]	$6 - 9 = -\ 3$ [NatM69]
$7 - 9 = -\ 2$ [NatM79]	$8 - 9 = -\ 1$ [NatM89]	$9 - 9 = 0$ [NatM99]

G.8 PrintNats

imports Layout$^{(G.10)}$ Nats$^{(G.7)}$
exports
  sorts PNAT
  lexical syntax
    [0-9]+ → PNAT
  context-free syntax
    "*print*"(Nat) → PNAT
hiddens
  variables
    nat[*0-9* ′]∗ → Nat
    ch[*0-9* ′]∗"∗"→ CHAR∗

equations

$$\frac{print(nat_1) = pnat(ch_1^*),\quad print(nat_2) = pnat(ch_2^*)}{print(nat_1\ nat_2) = pnat(ch_1^*\ ch_2^*)} \quad \text{[pr']}$$

$print(0) = pnat(\texttt{"0"})$ [pr0]

$print(1) = pnat(\texttt{"1"})$ [pr1]

$print(2) = pnat(\texttt{"2"})$ [pr2]

$print(3) = pnat(\texttt{"3"})$ [pr3]

$print(4) = pnat(\texttt{"4"})$ [pr4]

$print(5) = pnat(\texttt{"5"})$ [pr5]

$print(6) = pnat(\texttt{"6"})$ [pr6]

$print(7) = pnat(\texttt{"7"})$ [pr7]

$print(8) = pnat(\texttt{"8"})$ [pr8]

$print(9) = pnat(\texttt{"9"})$ [pr9]

G.9 Basics

imports Layout$^{(G.10)}$
exports
  sorts YN
  context-free syntax
    yes → YN
    no → YN
  variables
    Ch [*0-9* ′]∗ → CHAR
    Ch [*0-9* ′]∗"∗"→ CHAR∗
    "?" → YN

G.10 Layout

exports
  lexical syntax
    "%%"~[\n]∗[\n] → LAYOUT
    [␣\t\n\r] → LAYOUT

Appendix H

VA to C translation

The mapping below uses a number of macros (`preamble`, `external`, `constituent`, `entry` and `xfer`) that are used to define external functions for the interpreter. Other macros, like `dispatch`, are used to control the rewriting process in both the interpreter and the run-time system. Furthermore some special macros, like `..top`, `..psh`, `get_node`, etc., are used to access the machine stacks and heap. These macros are used instead of direct operations to hide the implementation of the μARM machine model for the translation process. This approach allows for easy changes in the implementation of the machine model. The preprocessor directive `##` is used to concatenate a macro parameter to a token to create a new macro or function call (for example `PUSH(AP)` expands to `APpsh`).

VA *instruction*	$\longmapsto$	*C code*

`DEFINE(Sym,FunName,FunVar,Id)`	$\longmapsto$	`fun FunName;`
		`function(FunName);` *and*
		`new_external(#Sym,FunName);`
		`varname=hdr2fun(lookup(#Sym));`

`FUNCTION(FunName)`	$\longmapsto$	`preamble(FunName)`
		`external FunName() {`
		   `constituent`
		   `entry(FunName);`

`END`	$\longmapsto$	`}`
`PUSH(Stack)`	$\longmapsto$	`Stack##psh`
`TOP(Stack)`	$\longmapsto$	`Stack##top`
`POP(Stack)`	$\longmapsto$	`Stack##pop(n)`
`MV(Val, Dest)`	$\longmapsto$	`Dest=(ref)Val;`
`NTH(Stack, N)`	$\longmapsto$	`Stack##nth(n)`
`DROP(Stack, N)`	$\longmapsto$	`Stack##ndrop(n);`

VA *instruction*	$\longmapsto$	*C instructions*
RECYCLE	$\longmapsto$	dispatch(IP); xfer;
GOTO(FunSym)	$\longmapsto$	IP = fun2crf(FunSym); xfer;
SWITCH	$\longmapsto$	Fun = funsym(APtop); {

CASE(ThisLabel, Sym, NextLabel)	$\longmapsto$	} if (Fun==Sym) {
		Ref=TPpsh=APpop;
		if (isref(Ref)) {
		if (Ref->cdr!=NUL) {
		APpsh=Ref->cdr;
		}
		APpsh=Ref->car;
		}

DEFAULT(ThisLabel)	$\longmapsto$	}
PAIRSKIP	$\longmapsto$	Ref = APtop; TPpsh = Ref; APtop = Ref->cdr; APpsh = Ref->car;
BUILD0(FunSym, Dest)	$\longmapsto$	Dest##psh=(ref)FunSym;
BUILD1(FunSym, Src, Dest)	$\longmapsto$	get_node(ref,Ref); Ref->fsym=FunSym; Ref->car=Src##pop; Ref->cdr=NUL; Dest##psh=Ref;
BUILD12(FunSym, Src, Dest)	$\longmapsto$	get_node(ref,Ref); Ref->fsym=FunSym; Ref->car=Src##pop; Ref->cdr=Src##pop; Dest##psh=Ref;
BUILD21(FunSym, Src, Dest)	$\longmapsto$	get_node(ref,Ref); Ref->fsym=FunSym; Ref->cdr=Src##pop; Ref->car=Src##pop; Dest##psh=Ref;

Appendix I

Example Translation

In this chapter we present an example of the translation of an ASF+SDF specification into C functions. The generated C functions can be compiled to object code and then linked to the run-time system or hybrid interpreter to create a stand-alone term rewriting system.

We will use the ASF+SDF specification of the data type of Numbers that is used for the measurements in Chapter 4 on page 21. It contains of a constant `z` denoting the number zero, a successor function, functions to add and multiply numbers and a function to compute the factorial of a number. All functions are written in prefix notation using respectively the function symbols `s`, `plus`, `times` and `fac`.

I.1 ASF+SDF specification

```
Module Numbers

exports
   sorts
      NUM
   lexical syntax
      "%%" ~[\n]* [\n]            -> LAYOUT
      [ \t\n]                     -> LAYOUT
   context-free syntax
      z                           -> NUM    %% z for zero
      s( NUM )                    -> NUM    %% s for successor
      plus( NUM , NUM )           -> NUM
      times( NUM , NUM )          -> NUM
      fac( NUM )                  -> NUM
   variables
      Num [0-9']*                 -> NUM

equations
   [N1]  plus( z, Num)            = Num
   [N2]  plus( s( Num1 ), Num2 )  = plus( Num1, s( Num2 ) )
   [N3]  times( z, Num)           = z
   [N4]  times( s( Num1 ), Num2 ) = plus( Num2, times( Num1, Num2 ) )
   [N5]  fac( z )                 = s( z )
   [N6]  fac( s( Num ) )          = times( s( Num), fac( Num ) )
```

I.2 μARM program

The front end generates an μARM program which contains a function definition for every function symbol that has rewrite rules (equations) in the ASF+SDF specification. Rewrite rules that concern the same function symbol are combined into one function definition.

```
start(
  fac,
  dock(
    match(
      s,
      skip(cpush(times,cpshv(1,apshv(0,tdrop(2,goto(fac)))))),
      match(
        z,
        cpshv(0,acbuild(s,1,tdrop(1,recycle))),
        skip(atbuild(fac,1,recycle))))),
  start(
    plus,
    dock(
      match(
        s,
        skip(skip(cpshv(0,ccbuild(s,1,c2a(apshv(1,
                                  tdrop(3,goto(plus)))))))),
        match(
          z,
          skip(cpshv(0,tdrop(2,recycle))),
          skip(skip(atbuild(plus,2,recycle)))))),
    start(
      z,
      acbuild(z,0,recycle),
      start(
        s,
        skip(cpshv(0,acbuild(s,1,tdrop(1,recycle)))),
        start(
          times,
          dock(
            match(
              s,
              skip(skip(cpush(plus,cpshv(0,apshv(0,apshv(1,
                                  tdrop(3,goto(times)))))))),
              match(
                z,
                skip(cpshv(1,tdrop(2,recycle))),
                skip(skip(atbuild(times,2,recycle)))))),
          notrie)))))
```

I.3 VA functions

The μARM program can be converted to a series of rewrite functions written in the Virtual Assembly language. The indentation shown in the functions below has no meaning. We have inserted it to increase readability.

```
FUNCTION( f_fac )
    SWITCH
        CASE( 101, v_s, 102 )
            MV( POP( AP ), PUSH( TP ) )
            MV( v_times, PUSH( CP ) )
            MV( NTH( TP, 1 ), PUSH( CP ) )
            MV( TOP( TP ), PUSH( AP ) )
            DROP( TP, 2 )
            GOTO( v_fac )
        CASE( 102, v_z, 103 )
            MV( TOP( TP ), PUSH( CP ) )
            BUILD1( v_s, CP, AP )
            DROP( TP, 1 )
            RECYCLE
        DEFAULT( 103 )
            MV( POP( AP ), PUSH( TP ) )
            BUILD1( v_fac, TP, AP )
            RECYCLE
END

FUNCTION( f_plus )
    SWITCH
        CASE( 101, v_s, 102 )
            MV( POP( AP ), PUSH( TP ) )
            MV( POP( AP ), PUSH( TP ) )
            MV( TOP( TP ), PUSH( CP ) )
            BUILD1( v_s, CP, CP )
            MV( POP( CP ), PUSH( AP ) )
            MV( NTH( TP, 1 ), PUSH( AP ) )
            DROP( TP, 3 )
            GOTO( v_plus )
        CASE( 102, v_z, 103 )
            MV( POP( AP ), PUSH( TP ) )
            MV( TOP( TP ), PUSH( CP ) )
            DROP( TP, 2 )
            RECYCLE
        DEFAULT( 103 )
            MV( POP( AP ), PUSH( TP ) )
            MV( POP( AP ), PUSH( TP ) )
            BUILD21( v_plus, TP, AP )
            RECYCLE
END

FUNCTION( f_z )
    BUILD0( v_z, AP )
    RECYCLE
END
```

```
FUNCTION( f_s )
    MV( POP( AP ), PUSH( TP ) )
    MV( TOP( TP ), PUSH( CP ) )
    BUILD1( v_s, CP, AP )
    DROP( TP, 1 )
    RECYCLE
END

FUNCTION( f_times )
    SWITCH
        CASE( 101, v_s, 102 )
            MV( POP( AP ), PUSH( TP ) )
            MV( POP( AP ), PUSH( TP ) )
            MV( v_plus, PUSH( CP ) )
            MV( TOP( TP ), PUSH( CP ) )
            MV( TOP( TP ), PUSH( AP ) )
            MV( NTH( TP, 1 ), PUSH( AP ) )
            DROP( TP, 3 )
            GOTO( v_times )
        CASE( 102, v_z, 103 )
            MV( POP( AP ), PUSH( TP ) )
            MV( NTH( TP, 1 ), PUSH( CP ) )
            DROP( TP, 2 )
            RECYCLE
        DEFAULT( 103 )
            MV( POP( AP ), PUSH( TP ) )
            MV( POP( AP ), PUSH( TP ) )
            BUILD21( v_times, TP, AP )
            RECYCLE
END
```

I.4 C functions

The VA functions are translated into portable C code using the Virtual Assembler and the C definitions file described in Appendix H on page 91.

```
preamble( f_fac )
external f_fac () {
        constituent

        entry( f_fac );

        Fun = funsym( APtop );
        if ( Fun == v_s ) {
                ReF = TPpsH = APpop;
                if ( isref( Ref ) ) {
                        if ( Ref->cdr != NUL ) {
                                APpsh = Ref->cdr;
                        }
                        APpsh = Ref->car;
                }
                StackCheck( PP <= TP );
                TPpsh = ( ref ) APpop;
```

```
                StackCheck( AP <= CP );
                CPpsh = ( ref ) v_times;
                StackCheck( AP <= CP );
                CPpsh = ( ref ) TPnth( 1 );
                StackCheck( AP <= CP );
                APpsh = ( ref ) TPtop;
                TPndrop( 2 );
                IP = fun2crf( v_fac );
                xfer;
        }
        if ( Fun == v_z ) {
                ReF = TPpsH = APpop;
                if ( isref( Ref ) ) {
                        if ( Ref->cdr != NUL ) {
                                APpsh = Ref->cdr;
                        }
                        APpsh = Ref->car;
                }
                StackCheck( AP <= CP );
                CPpsh = ( ref ) TPtop;
                get_node( ref, Ref );
                Ref->fsym = v_s;
                Ref->car = CPpop;
                Ref->cdR = NUL;
                APpsH = Ref;
                TPndrop( 1 );
                dispatch( IP );
                xfer;
        }
        StackCheck( PP <= TP );
        TPpsh = ( ref ) APpop;
        get_node( ref, Ref );
        Ref->fsym = v_fac;
        Ref->car = TPpop;
        Ref->cdR = NUL;
        APpsH = Ref;
        dispatch( IP );
        xfer;
}


preamble( f_plus )
external f_plus () {
        constituent

        entry( f_plus );

        Fun = funsym( APtop );
        if ( Fun == v_s ) {
                ReF = TPpsH = APpop;
                if ( isref( Ref ) ) {
                        if ( Ref->cdr != NUL ) {
                                APpsh = Ref->cdr;
```

```
                    }
                    APpsh = Ref->car;
            }
            StackCheck( PP <= TP );
            TPpsh = ( ref ) APpop;
            StackCheck( PP <= TP );
            TPpsh = ( ref ) APpop;
            StackCheck( AP <= CP );
            CPpsh = ( ref ) TPtop;
            get_node( ref, Ref );
            Ref->fsym = v_s;
            Ref->car = CPpop;
            Ref->cdR = NUL;
            CPpsH = Ref;
            StackCheck( AP <= CP );
            APpsh = ( ref ) CPpop;
            StackCheck( AP <= CP );
            APpsh = ( ref ) TPnth( 1 );
            TPndrop( 3 );
            IP = fun2crf( v_plus );
            xfer;
        }
        if ( Fun ==  v_z ) {
            ReF = TPpsH = APpop;
            if ( isref( Ref ) ) {
                    if ( Ref->cdr != NUL ) {
                            APpsh = Ref->cdr;
                    }
                    APpsh = Ref->car;
            }
            StackCheck( PP <= TP );
            TPpsh = ( ref ) APpop;
            StackCheck( AP <= CP );
            CPpsh = ( ref ) TPtop;
            TPndrop( 2 );
            dispatch( IP );
            xfer;
        }
        StackCheck( PP <= TP );
        TPpsh = ( ref ) APpop;
        StackCheck( PP <= TP );
        TPpsh = ( ref ) APpop;
        get_node( ref, Ref );
        Ref->fsym = v_plus;
        Ref->cdr = TPpop;
        Ref->car = TPpop;
        APpsH = Ref;
        dispatch( IP );
        xfer;
}


preamble( f_z )
```

```
external f_z () {
        constituent

        entry( f_z );

        APpsh = ( ref ) v_z;
        dispatch( IP );
        xfer;
}


preamble( f_s )
external f_s () {
        constituent

        entry( f_s );

        StackCheck( PP <= TP );
        TPpsh = ( ref ) APpop;
        StackCheck( AP <= CP );
        CPpsh = ( ref ) TPtop;
        get_node( ref, Ref );
        Ref->fsym = v_s;
        Ref->car = CPpop;
        Ref->cdR = NUL;
        APpsH = Ref;
        TPndrop( 1 );
        dispatch( IP );
        xfer;
}


preamble( f_times )
external f_times () {
        constituent

        entry( f_times );

        Fun = funsym( APtop );
        if ( Fun == v_s ) {
                ReF = TPpsH = APpop;
                if ( isref( Ref ) ) {
                        if ( Ref->cdr != NUL ) {
                                APpsh = Ref->cdr;
                        }
                        APpsh = Ref->car;
                }
                StackCheck( PP <= TP );
                TPpsh = ( ref ) APpop;
                StackCheck( PP <= TP );
                TPpsh = ( ref ) APpop;
                StackCheck( AP <= CP );
                CPpsh = ( ref ) v_plus;
```

```
                StackCheck( AP <= CP );
                CPpsh = ( ref ) TPtop;
                StackCheck( AP <= CP );
                APpsh = ( ref ) TPtop;
                StackCheck( AP <= CP );
                APpsh = ( ref ) TPnth( 1 );
                TPndrop( 3 );
                IP = fun2crf( v_times );
                xfer;
        }
        if ( Fun == v_z ) {
                ReF = TPpsH = APpop;
                if ( isref( Ref ) ) {
                        if ( Ref->cdr != NUL ) {
                                APpsh = Ref->cdr;
                        }
                        APpsh = Ref->car;
                }
                StackCheck( PP <= TP );
                TPpsh = ( ref ) APpop;
                StackCheck( AP <= CP );
                CPpsh = ( ref ) TPnth( 1 );
                TPndrop( 2 );
                dispatch( IP );
                xfer;
        }
        StackCheck( PP <= TP );
        TPpsh = ( ref ) APpop;
        StackCheck( PP <= TP );
        TPpsh = ( ref ) APpop;
        get_node( ref, Ref );
        Ref->fsym = v_times;
        Ref->cdr = TPpop;
        Ref->car = TPpop;
        APpsH = Ref;
        dispatch( IP );
        xfer;
}
```